

Maciej Gil

Inżynieria oprogramowania

Warszawa 2012



KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY



Politechnika Warszawska
Wydział Samochodów i Maszyn Roboczych
Kierunek studiów "Edukacja techniczno informatyczna"
02-524 Warszawa, ul. Narbutta 84, tel 22 849 43 07, 22 234 83 48
ipbmvr.simr.pw.edu.pl/spin/, e-mail: sto@simr.pw.edu.pl

Opiniodawca: prof. dr hab. inż. Jerzy WRÓBEL

Projekt okładki: Norbert SKUMIAŁ, Stefan TOMASZEK

Projekt układu graficznego tekstu: Grzegorz LINKIEWICZ

Skład tekstu: Magdalena BONAROWSKA

Publikacja bezpłatna, przeznaczona dla studentów kierunku studiów
"Edukacja techniczno-informatyczna".

Copyright © 2012 Politechnika Warszawska

Utwór w całości ani we fragmentach nie może być powielany
ani rozpowszechniany za pomocą urządzeń elektronicznych, mechanicznych,
kopiujących, nagrywających i innych bez pisemnej zgody posiadacza praw
autorskich.

ISBN 83-89703-98-X

Druk i oprawa: STUDIO MULTIGRAF sp. z o.o.,
ul. Ołowiana 10, 85-461 Bydgoszcz

Spis treści

Wstęp	5
1. Wprowadzenie.....	7
2. Cykl „życia” oprogramowania	13
2.1. Model: twórz kod i poprawiaj	15
2.2. Model: kaskadowy	16
2.3. Model: przyrostowy	18
2.4. Model: prototypowy	21
2.5. Model: spiralny	22
2.6. Inne modele	24
3. Specyfikacja wymagań w procesie tworzenia oprogramowania	29
4. Zagadnienia analizy strukturalnej i projektowanie.....	33
4.1. Wprowadzenie	34
4.2. Modelowanie zorientowane na procesy	35
4.3. Modelowanie zorientowane na dane	36
4.4. Modelowanie obiektowe.....	38
4.5. UML 2.0	43
4.6. Generowanie modeli UML w narzędziach Enterprise Architect.....	46
4.7. Generowanie kodu źródłowego i inżynieria wsteczna	49
4.8. Generowanie Dokumentacji w Enterprise Architect	50
5. Zarządzanie budową oprogramowania	53
5.1. Wprowadzenie	54
5.2. Generacje języków programowania	57
5.3. Cechy języków programowania	60
5.4. Języki skryptowe	62
5.5. Programowanie proceduralne i deklaratywne	63
5.6. Obiektowe języki programowania.....	64

5.7. Re - używalność kodu	66
5.8. Aplikacje działające w środowiskach rozproszonych	67

6. Narzędzia wykorzystywane w inżynierii oprogramowania..... 69

6.1. Komputerowo wspomagana inżynieria oprogramowania CASE.....	70
6.2. Zintegrowane środowiska deweloperskie.....	72

7. Kontrola i monitorowanie w inżynierii oprogramowania 75

7.1. Diagram Gantta	76
7.2. Harmonogramowanie PERT/CPM.....	77
7.3. Problemy z harmonogramowaniem projektów.....	80

8. Jakość Oprogramowania 83

8.1. Inspekcja i walidacja	84
8.2. Zarządzanie jakością oprogramowania.....	89

Bibliografia 93

Wstęp

Niniejsza książka została opracowana jako materiał dydaktyczny w ramach realizacji Programu Rozwojowego Politechniki Warszawskiej, współfinansowanego ze środków PROGRAMU OPERACYJNEGO KAPITAŁ LUDZKI. Przeznaczona jest dla studentów trzeciego roku studiów inżynierskich, na kierunku nauczania „Edukacja techniczno-informatyczna”, na Wydziale Samochodów i Maszyn Roboczych Politechniki Warszawskiej.

Swoim zakresem obejmuje zagadnienia określone w programie studiów dla przedmiotu pod tym samym tytułem. Program tego przedmiotu, jak również zakres książki, jest zgodny ze standardami kształcenia dla kierunku nauczania „Edukacja techniczno-informatyczna” na studiach pierwszego stopnia w zakresie informatyki i systemów informatycznych, określonymi przez Ministerstwo Nauki i Szkolnictwa Wyższego.

„Inżynieria oprogramowania” to kolejny przedmiot z grupy przedmiotów informatycznych, przeznaczony dla studentów trzeciego roku studiów. Równolegle, na piątym semestrze studenci kierunku nauczania „edukacja techniczno-informatyczna”, mają inne zajęcia z przedmiotu należącego do grupy przedmiotów informatycznych: „programowanie i programy użytkowe”, który jest uzupełnieniem i rozwinięciem zakresu niniejszej pracy. Dla studentów dostępnych jest jeszcze wiele przedmiotów przybliżających ważne zagadnienia technologii informatycznych, wśród nich takie jak: „Sieci komputerowe i aplikacje sieciowe”, „Techniki multimedialne w dydaktyce” jak również wiele przedmiotów obieralnych, a także praca przejściowa i praca dyplomowa.

Głównym celem tej pracy jest zapoznanie studentów z podstawowymi pojęciami związanymi z inżynierią oprogramowania. Praca składa się z 8 rozdziałów. Po wprowadzeniu omówiono podstawowe modele tzw. cyklu „życia” oprogramowania. W rozdziale kolejnym dokonano analizy najważniejszych wymagań formułowanych w procesie tworzenia oprogramowania. W rozdziale czwartym omówiono podstawowe metody modelowania przyjmowane w procesie tworzenia oprogramowania. Przeanalizowano modelowanie zorientowane na procesy, modelowanie zorientowane na dane, modelowanie obiektowe oraz omówiono podstawy języka UML stosowanego w procesie tworzenia oprogramowania. Rozdział piąty to przegląd pojęć i technik stosowanych w procesie tworzenia oprogramowania. W rozdziale szóstym dokonano przeglądu narzędzi stosowanych w procesie tworzenia kodu oprogramowa-

nia. Rozdziały siódmy i ósmy poświęcone są kolejno harmonogramowaniu i kontroli jakości oprogramowania.



Wprowadzenie

Oprogramowanie komputerowe w większości przypadków realizowane jest w postaci projektów informatycznych. Projekty, w różnej formie, od dawna związane są z działalnością człowieka. Już od najdawniejszych czasów człowiek organizował się w grupy w celu współdziałania nawet przy prostych czynnościach jak zbieractwo, uprawy czy polowania. Już wtedy inteligencja człowieka pozwoliła dostrzec korzyści płynące z usystematyzowania i wykonywania pewnych czynności w sposób zorganizowany. W trakcie rozwoju cywilizacji, przedsięwzięcia podejmowane przez ludzi stawały się coraz bardziej skomplikowane i wyrafinowane. Podejmowanie takich przedsięwzięć wiązało się a koniecznością ich uprzedniego zaplanowania a następnie kierowania grupą osób zaangażowanych w ich wykonywanie. Przedsięwzięcia zawsze wiązały się z ustaleniem celu działania oraz określeniu sposobu wykonania poszczególnych celów ze względu na zdefiniowane ograniczenia. Mogły być to ograniczenia związane z zasobami, kosztami, czasem, posiadanymi narzędziami, kadrą i wiele innych. W ostatnim czasie, głównie w branży budowlanej, przemyśle oraz w informatyce, projekty stały się na tyle zawansowane a jednocześnie ich skala i znaczenie wzrosły na tyle, że spowodowało to zapotrzebowanie na zajęcie się tematyką projektowania samą w sobie, jako dziedzinę poznania, dziedziną nauki. Zarządzanie projektami stało się odrębną profesją popartą naukową wiedzą. Współcześnie istnieje szereg niezależnych organizacji zajmujących się tą tematyką, w tym wiele organizacji non-profit i międzynarodowych. Wiodącą jest organizacja (PMI, 2008) (ang.: Project Management Institute) (zobacz: <http://www.pmi.org>). Jest to niezależna organizacja o międzynarodowym charakterze, zajmująca się tworzeniem ogólnych norm, wytycznych i zasad obowiązujących w zarządzaniu dowolnymi projektami. W obszarze jej działalności mieści się również tworzenie obowiązujących standardów oraz certyfikacje, które uznawane są w każdej branży.

Według PMI, projekt to każde przedsięwzięcie o zdefiniowanym punkcie początkowym oraz ze zdefiniowanymi celami definiującymi zakończenie. Każdy projekt zależy od ograniczonych zasobów, za pomocą których cele te mogą zostać zrealizowane (Project Management Institute, 2001).

PMI powstał na początku lat 70-tych ubiegłego stulecia i od samego początku jest uważany, za najbardziej uznaną organizację specjalistyczną w dziedzinie zarządzania projektami. W skład tej organizacji zaangażowana jest międzynarodowa grupa fachowców profesjonalnie zajmujących się zarządzaniem dużymi projektami. Ich głównym zadaniem jest określanie standardów nowoczesnego zarządzania projektem. Instytut ten podaje najbardziej ogólną definicję zarządzania projektem jako:

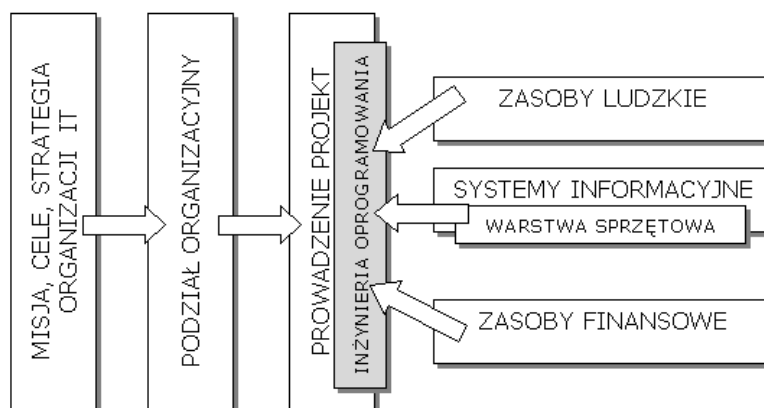
Zarządzanie projektem to sztuka kierowania i koordynowania ludzkimi oraz materialnymi zasobami w trakcie realizacji przedsięwzięcia, za pomocą technik nowoczesnego zarządzania w celu osiągnięcia zdefiniowanych celów co do kosztów, czasu, jakości i satysfakcji z wykonania tego przedsięwzięcia.

PMI opublikował wytyczne, które stały się podstawą obowiązującej normy ANSI. Wytyczne te zawarte zostały w pracy (PMI, 2008). Jak wspomniano wcześniej, w przeważającej większości, tworzenie oprogramowania jest związane z przeprowadzeniem projektu. Jest to jeden z kluczowych aspektów całej współczesnej branży technologii informatycznych. Jak pokazują wyniki finansowe wielu przedsiębiorstw, działalność związana z tworzeniem i wdrażaniem oprogramowania jest jedną z najistotniejszych pozycji w budżecie działów IT. Jednocześnie, szczególnie w świecie tworzenia oprogramowania, szczególnie w przedsięwzięciach mniejszej skali, zauważyć można tendencję do umniejszania i ignorowania znaczenia zarządzania projektem, co w efekcie kończy się niepowodzeniem takich przedsięwzięć, związanym głównie z opóźnieniami dotyczącymi rozwoju oprogramowania oraz znacznym przekraczaniem kosztów realizacji. Często zdarza się również, że projekty związane z budową oprogramowania kończą się fiaskiem z podstawowego powodu, to znaczy nie spełniają one oczekiwań użytkowników końcowych - są z ich punktu widzenia nieużyteczne.

Według Standish Group (J. Laurenz Eveleens, 2010) jedynie 1/3 wszystkich projektów w dziedzinie tworzenia oprogramowania kończy się sukcesem. Pozostałe 2/3 to projekty złe, a więc nieukończone, znacząco przekraczające założony budżet lub uznanych za niepomyślnie zakończonych – czytaj wyrzuconych do kosza. Za optymistyczny można uznać jednak fakt, że zaobserwować można trwały trend do poprawiania tych statystyk. Z roku na rok odsetek dobrych projektów jest coraz większy. Według tych autorów, wiele wskazuje na to, że poprawa powodowana jest wzrostem świadomości co do sposobu realizacji projektów informatycznych.

Biorąc pod uwagę całą branżę IT, projekty związane ściśle z inżynierią oprogramowania zlecane są i podejmowane przez powołane do tego celu jednostki organizacyjne w celu realizacji odpowiedniej strategii biznesowych. Projekty z kolei kształtują procesy zachodzące w całej firmie. Do realizacji wszystkich tych warstw niezbędne są odpowiednie zasoby ludzkie oraz odpowiednie zasoby finansowe. W efekcie następowania tych tendencji można zauważyć, że w ostatnich latach stają się coraz bardziej niezbędne, powstałe systemy informatyczne, które dodatkowo

oparte są na odpowiedniej infrastrukturze (komputerowej oraz sieciowej). Zależności te przedstawiono na schemacie 1.



Rysunek 1. Inżynieria oprogramowania w relacji do zadań biznesowych firmy oraz wykorzystania jej zasobów

Tworzenie oprogramowania może być zdefiniowane jako zbiór działań wykonywanych w oparciu o zestaw metod i praktyk, których celem jest stworzenie oprogramowania oraz produktów z nim związanych. Można wyobrazić sobie pewną analogię pomiędzy projektem konstrukcyjnym a projektem programistycznym. Chociaż istnieje wiele podobieństw pomiędzy tymi dwoma typami projektów, to jednak pojawiają się także elementy, które są charakterystyczne tylko dla danej branży.

Projekt konstrukcyjny jest zwykle bardziej autonomiczny, dotyczy pewnej zamkniętej całości, części, zespołu lub maszyny, w efekcie w jego postęp zaangażowane są zwykle projektanci, lub zespół projektowy w ramach pojedynczej jednostki organizacyjnej. Projekt programistyczny wkomponowany jest w infrastrukturę informatyczną całej firmy, przez co jest on bardziej rozproszony i jednocześnie ma na niego wpływ więcej osób będących przedstawicielami różnych działów w całej firmie. Ponadto, zawsze warunkiem działania nowego oprogramowania jest ciągłość, to znaczy jego zgodność z innymi systemami lub oprogramowaniem już zainstalowanym i działającym w firmie, bądź też innym oprogramowaniem, z którego korzystają pracownicy. Weźmy przykładowo oprogramowanie CAD. Jeżeli nowy program CAD nie będzie wspierał formatów zapisu danych to na pewno nie zostanie on zaakceptowany i przyjęty jako obowiązujący. Jeżeli nowy program CAD w sposób znaczący zmieni interfejs użytkownika, to również może nie zostać przyjęty pozytywnie. W przypadku projektów informatycznych należy więc za-

pewnie pewną ciągłość i kompatybilność zarówno z systemami jak i użytkownikami końcowymi. Projekty konstrukcyjne nie wymagają już takiej kompatybilności. Wprowadzenie nowego modelu samochodu nie oznacza że musi on współpracować z poprzednim, czy być do poprzednika podobny – a wręcz w przypadku samochodów – różnice są zjawiskiem pożądanym ze względów marketingowych.

Głębsze porównanie obu rodzajów projektów pokazuje również, że znalezienie dodatkowych zasobów ludzkich, w przypadku gdy zachodzi taka potrzeba, jest łatwiejsze w przypadku projektu konstrukcyjnego. Wdrożenie inżyniera do projektu IT z reguły wymaga dużo dłuższego czasu i jego początkowy wkład jest z reguły mało produktywny. Różnice te mogą wynikać z natury organizacyjnej obu rodzajów przedsięwzięć: w projektach konstrukcyjnych priorytetem jest zwykle czas realizacji – termin oddania nowego budynku, czas wprowadzenia nowego modelu samochodu, itd.. W projektach programistycznych, termin zakończenia ustalany jest z góry (tak zwany dead-line), a następnie główny nacisk kładziony jest na zasoby ludzkie – a te są zwykle ograniczane.

Kolejna różnica dotyczy pomiaru postępów. W przypadku projektów informatycznych mierzenie postępów nie jest łatwe. W realizacji projektu konstrukcyjnego zwykle zaczynamy od tworzenia kolejnych części, potem złożań, w końcu całej maszyny - postęp prac jest zauważalny, niemalże namacalny. W projektach IT zdecydowanie trudniej wyróżnić i dostrzec kolejne efekty pracy. W zasadzie działające oprogramowanie można zobaczyć i przetestować dopiero w końcowej fazie projektu. Jest to często taki moment, gdy za późno jest już na poprawki, a te które okazują się niezbędne do wprowadzenia, znacznie podwyższają budżet.

Podsumowując, projekty inżynierii oprogramowania różnią się projektów konstrukcyjnych, a to pociąga za sobą konieczność stosowania nieco odmiennego podejścia do sterowania nimi oraz odmiennych metod i narzędzi do ich prowadzenia. Zrozumienie i zapoznanie się z nimi jest niezbędne do takiego prowadzenia projektów aby zakończyły się one sukcesem. Nawiązując do wcześniej przytoczonych statystyk, należy robić wszystko aby znaleźć się w tej 1/3 projektów zakończonych sukcesem – jednocześnie poprawiając tę statystykę.

2

Cykl „życia” oprogramowania

W poprzednim rozdziale zwrócono uwagę na istotę systematycznego podejście do tworzenia oprogramowania poprzez wykorzystanie nauki o prowadzeniu projektów, jako ściśle zorganizowanego procesu. Należy jednak zauważyć, że samo tworzenie oprogramowania jest samo w sobie elementem większej układanki, która określana jest mianem całego cyklu życia oprogramowania.

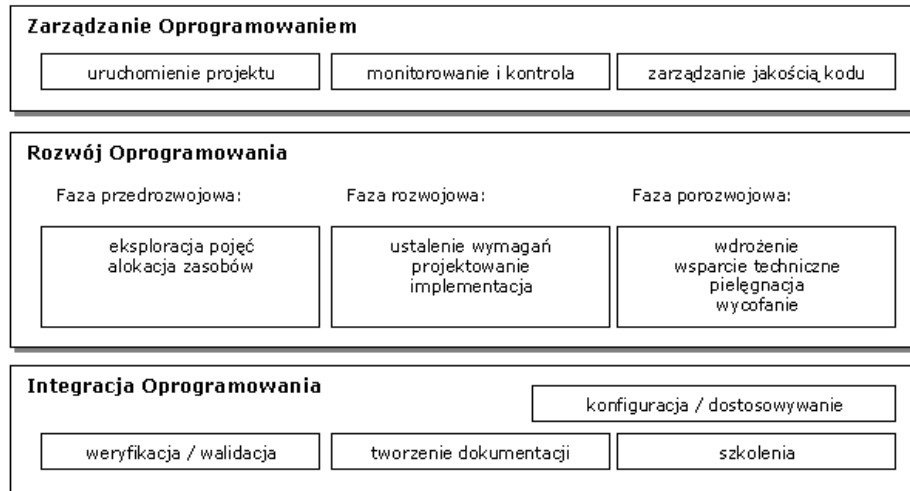
Oprogramowanie jak każdy wytwór ludzki podlega pewnemu cyklowi. Określony on jest więc w ramach czasowych. Punktem początkowym jest pomysł na oprogramowanie jako pewien system informatyczny. Punktem końcowym jest moment, w którym produkt jest wdrożony i udostępniony do użytku. Często jednak zdarza się, że cykl życia rozciąga się dalej i dotyczy również utrzymania oprogramowania oraz wprowadzanie zmian czyli nowych wersji. Ogólnie, można więc wyróżnić następujące fazy życia oprogramowania:

- formułowanie wymagań
- projektowanie
- implementacja
- testowanie
- wdrażanie
- często: eksploatacja i utrzymanie
- czasem: wycofanie z eksploatacji

Model cyklu życia oprogramowania jest pewną abstrakcją, za pomocą której możemy opisać a przez to również zarządzać, rozwojem tego oprogramowania. Cykl życia oprogramowania jest często nazywany cyklem rozwoju systemu - (ang: Software Development Life Cycle). Działania i procesy potrzebne do tworzenia i pielęgnacji oprogramowania również zostały poddane standaryzacji i zostały opisane w postaci normy IEEE 1074. Zakłada ona określone ramy tworzenia modelu cyklu życia. Według tego standardu wyróżnić możemy pewne grupy procesów, które stanowią punkty wyjścia do budowy konkretnych modeli cyklu życia. Procesy te ilustruje Rysunek 2.

Najbardziej liczne i jednocześnie najbardziej rozbudowane procesy występują a fazach związanych z rozwojem oprogramowania. Wchodzą one jednocześnie w obszar zainteresowania inżynierii oprogramowania. Procesy te doczekały się wypracowania szeregu różnych modeli, które zo-

stały dobrze przetestowane, opisane i sformalizowane. Ich przydatność w praktyce jest niepodważalna. Często mówi się o nich jako o metodologii rozwoju oprogramowania. W dalszej części rozdziału zaprezentowano i omówiono najpopularniejsze, współcześnie stosowane metodologie (cykle) związane z inżynierią rozwoju oprogramowania.



Rysunek 2. Procesy składowe cyklu życia oprogramowania

2.1 Model: twórz kod i poprawiaj

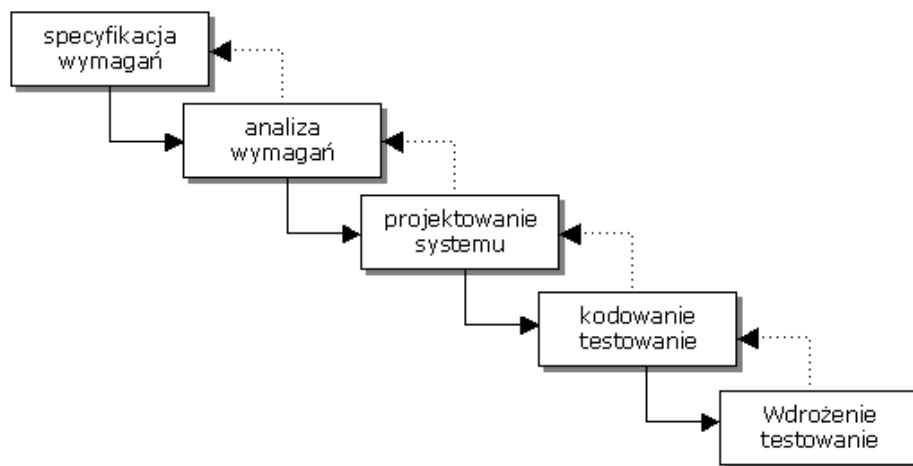
Podejście do tworzenia oprogramowania poprzez szybkie pisanie kodu i poprawianie (ang.: code-and-fix) jest najprostszym modelem rozwoju oprogramowania. Próżno w nim szukać fazy planowania, specyfikacji i projektowania. Tworzenie oprogramowania zaczyna się od razu od pisania kodu, często na szybko, na 'wyczucie' oraz zgrubnie (ang.: quick-and-dirty). W następnym kroku następuje kompilacja, a dalej od razu testowanie i wprowadzanie niezbędnych poprawek, aż do uzyskania zadowalającego rezultatu.

Tego rodzaju podejście, oparte na metodzie prób i błędów zwykle sprawdza się w przypadku małych, najczęściej dość autonomicznych przedsięwzięć, które mogą narodzić się w głowie jednej osoby i mogą

przez tę osobę zostać w całości zrealizowane. Jest to podejście typowe i jak najbardziej racjonalne w tego typu mini-projektach. W przypadku budowy bardziej złożonego oprogramowania – ta metoda zupełnie się nie sprawdzi.

2.2 Model: kaskadowy

Model kaskadowy (ang.: waterfall) jest jednym z pierwszych modeli rozwoju, powstałym jeszcze na początku lat 70-tych XX wieku. Polega on na realizacji procesów w określonym porządku, jeden po drugim. Rozwój następuje więc stopniowo, sekwencyjnie, tak jak na pokazuje to Rysunek 3, na którym przedstawiono schemat rozwoju oprogramowania według modelu kaskadowego.



Rysunek 3. Schemat modelu kaskadowego rozwoju oprogramowania

Model kaskadowy jest zwykle stosowany w następujących sytuacjach:

- dziedzina projektu jest dobrze znana nie tylko zleceniodawcy ale także jest dobrze znana zespołowi projektowemu,
- dziedzina projektu nie ewoluje w okresie jego realizacji,
- zleceniodawca ma możliwość czekania z wdrożeniem aż do czasu przygotowania gotowego rozwiązania.

Model kaskadowy jest najbardziej podobnym do podejścia stosowanego w projektach konstrukcyjnych. Jak wiemy jednak, istnieją pewne różnice między projektami IT a projektami ściśle technicznymi, przez co model kaskadowy nie uwzględnia wszystkich cech projektowania oprogramowania. Na przykład, projekty mechaniczne zawsze w na pewnym etapie wymagają produkcji fabrycznej, seryjnej, czego nie można spotkać w projektach informatycznych, z których każdy jest znacząco różny od poprzedniego. Realizacja projektu konstrukcyjnego może być porównana do budowy domu, zaś tworzenie oprogramowania do rzeźby. Rzeźba jest zajęciem mniej umocowanym, bardziej plastycznym, formowany obiekt powstaje przez dodawanie, ujmowanie materiału w celu nadania zamierzonego kształtu. Budowa domu jest bardziej sformalizowana i poddana obiektywnym a nie subiektywnym kryteriom. W efekcie zastosowanie tego rodzaju modelu do projektów informatycznych z jednej strony upraszcza zarządzanie i ułatwia planowanie przedsięwzięcia; z drugiej jednak strony, narzucenie ścisłej kolejności prac usztywnia twórcze elementy procesu.

Ogólnie można powiedzieć, że w modelu kaskadowym realnie istnieje możliwość powrotu do działania bezpośrednio poprzedzającego na zasadzie pętli zwrotnej (ang. feedback loop). Taka sytuacja może mieć miejsce, jeżeli w trakcie aktualnego działania nastąpi jakiś istotny problem bądź niespójność uniemożliwiająca dalszy postęp prac. Należy zwrócić również uwagę na koszty związane z popełnieniem ewentualnego błędu. Jeżeli jakiś błąd nastąpił już we wcześniejszej fazie, wycofanie staje się o wiele bardziej kosztowne, gdyż musimy cofać się również kaskadowo, krok po kroku, a następnie iść nowym torem jeszcze raz w sposób kaskadowy. Powoduje to narażenie projektu na duże straty w wymiarze czasowym oraz finansowym.

Bardziej zaawansowana odmiana modelu kaskadowego wprowadza na każdym etapie, na każdej kaskadzie elementy testowania lub walidacji - oczywiście na danym poziomie abstrakcji. W takim podejściu, każda kolejna kaskada powoduje uszczegółowienie projektu aż do pojedynczej linii kodu. Na każdym etapie dokonuje się testu lub walidacji w stosunku do poprzedniego kroku (kaskady położonej bezpośrednio nad bieżącą).

Model kaskadowy może więc być przydatny w sytuacjach gdy dziedzina jest dobrze znana i dobrze zrozumiana przez wszystkich uczestników przedsięwzięcia oraz jest ona relatywnie stabilna, a więc nie zmienia się dynamicznie w czasie, gdyż w przeciwnym przypadku mogą powstać problemy trudne do przezwyciężenia, polegające na niemożności precyzyjnego zdefiniowania wszystkich wymagań w fazie początkowej projektu. Jeżeli następować miałyby następować zmiany wywołane czynni-

kami zewnętrznymi, to zaistniałyby poważne trudności w ich adaptacji w projekcie. Jak się często okazuje w praktyce, rzeczywiste, duże projekty programistyczne rzadko kiedy mogą być przeprowadzone w tak stabilnych warunkach, przez co model ten nie jest skalowalny do tego rodzaju przedsięwzięć, a więc stosowanie modelu kaskadowego jest nieracjonalne.

Należy zwrócić tu uwagę na kolejny ważny element. W modelu kaskadowym nie ma możliwości dostarczenia wersji próbnej systemu, tzw. demo systemu. Funkcjonowanie systemu może być praktycznie uwidocznione dopiero w końcowych fazach projektu. Jest to czynnik in minus, gdyż jest on obciążony dużym ryzykiem w postaci poniesienia znacznych kosztów na projekt, który w ostatniej fazie może uwidocznienie jakieś istotne dla odbiorcy lub użytkownika braki - a ich poprawa wiąże się z kosztownymi ruchami „pod prąd”. Z uwagi na opisane tutaj słabe strony modelu kaskadowego powinno się ograniczać jego użycie do sytuacji, w których wymagania i ich implementacja są dobrze zrozumiane, sformalizowane i przestrzegane przez wszystkie strony uczestniczące w tworzeniu oprogramowania. Na przykład, jeżeli dany zespół osób ma doświadczenie w tworzeniu dedykowanych aplikacji CAD, to może on w swoich kolejnych projektach stosować model kaskadowy, którego szkielet został wypracowany i sprawdzony na bazie poprzednich projektów.

2.3 Model: przyrostowy

Dalszym rozwinięciem modelu kaskadowego jest sposób budowy oprogramowania według modelu przyrostowego (ang.: incremental development). Zasadniczo model ten realizuje on te same założenia co poprzednik, ale jest on wyraźnie podzielony na mniejsze części, które dodatkowo zachodzą na siebie. Zabieg ten pozwala skrócić cały proces, jednocześnie dając odbiorcy gotowego systemu możliwość wglądu w jego poszczególne elementy funkcjonalności już we wcześniejszych etapach cyklu życia projektu. Podejście do tworzenia modelu przyrostowego zakłada następujące czynności:

- specyfikacja wymagań – jest to specyfikacja wstępna, przygotowana na tyle, na ile jest to możliwe na tym etapie
- wykonanie wstępnego, bardzo ogólnego projektu całości

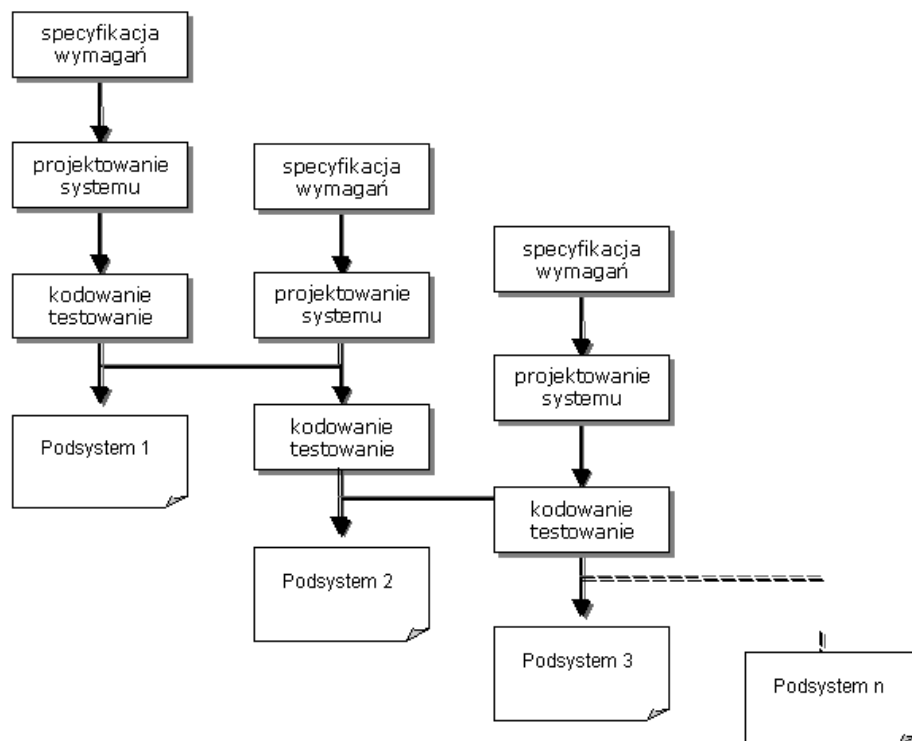
- wybór pewnego podzbioru funkcjonalnego systemu, tzw. podsystemu.
- szczegółowy projekt podsystemu, tutaj już według modelu kaskadowego
- implementacja podsystemu realizującego zadaną funkcjonalność
- testowanie zrealizowanego fragmentu i przedstawienie go zleceniodawcy celem weryfikacji i walidacji
- wybór kolejnego podsystemu obejmującego wybrany zakres funkcji
- powtarzanie kolejnych etapów, aż do implementacji całego systemu

W sytuacjach, kiedy jedne moduły są implementowane przed innymi, wymagane jest precyzyjne zdefiniowanie interfejsów pomiędzy podsystemami, co jest warunkiem ich późniejszej interoperacyjności. Tok postępowania przy tworzeniu podsystemów w modelu przyrostowym pokazuje Rysunek 4.

Wśród zalet takiej dekompozycji cyklu życia oprogramowania należy wymienić możliwość częstych, produktywnych kontaktów z odbiorcą. Każdy wyłoniony podsystem może być konsultowany o wiele wcześniej w porównaniu do możliwości pokazywania efektów projektu w modelu kaskadowym. Nie ma również konieczności definiowania z góry, i przestrzegania całości wymagań, co było koniecznością w modelu kaskadowym. W pierwszym etapie dokonuje się tylko specyfikacji wymagań wstępnych. Wymagania szczegółowe doprecyzowywane są w momencie realizacji projektu dla danego podsystemu. Jak widać, prace projektowe można do pewnego stopnia zrównoleglić, co ma wpływ na uelastycznienie całego procesu, a pewne potencjalne opóźnienia w rozwoju jednego podsystemu mają minimalny, bądź żaden, wpływ na rozwój pozostałych podsystemów. Ponadto, często okazuje się, że wybrane podsystemy, jeżeli są już gotowe, mogą być praktycznie wykorzystane przez odbiorcę jeszcze przed formalnym zakończeniem całości projektu, co często jest bardzo atrakcyjnym rozwiązaniem dla klienta, gdyż może on założyć sobie odpowiedni harmonogram dostosowania się i wdrażania nowego oprogramowania.

Przy stosowaniu modelu przyrostowego należy pamiętać że choć podział cyklu na wybrane podsystemy funkcjonalne ma niewątpliwe zalety przy-

toczone wcześniej, to również z drugiej strony takie rozwiązanie jest obciążone dodatkowymi kosztami związanymi z niezależną realizacją fragmentów (podsystemów). W wielu rzeczywistych przypadkach mogą również pojawiać się trudności z podziałem przedsięwzięcia na niezależne podsystemy funkcjonalne gdyż granice tych podsystemów nie są ewidentne, a ewentualne błędy tego podziału mogą ujawnić się w kolejnych fazach generując dodatkowe koszty projektu.



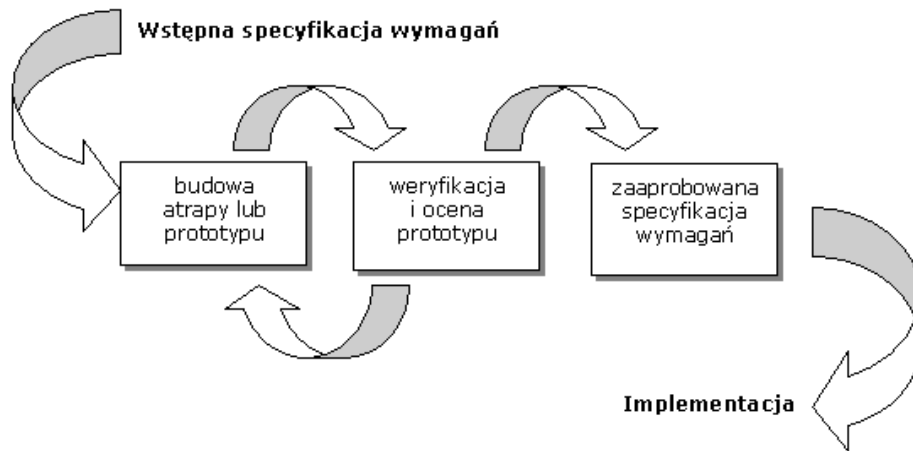
Rysunek 4. Schemat modelu przyrostowego rozwoju oprogramowania

Jeżeli projekt jest duży, ale można w nim wskazać podsystemy, to należy wskazać takie elementy, bez których system mógłby działać z okrojoną funkcjonalnością, to można w takim przypadku dokonać podziału na podsystemy a następnie bezpiecznie zastosować model przyrostowy. Model przyrostowy będzie więc sprawdzał się tam, gdzie nieracjonalne jest stosowanie modelu kaskadowego, a więc w takich sytuacjach, w których wytworzenie całego systemu 'na raz' jest obciążone zbyt dużym ryzykiem.

2.4 Model: prototypowy

Model prototypowy tworzenia oprogramowania polega na tym, że już na etapie projektowania tworzona jest atrapa (prototyp) oprogramowania w celu przedyskutowania i uzyskania akceptacji u odbiorcy. Dopiero po takich uzgodnieniach następuje realizacja kolejnych etapów tworzenia oprogramowania.

Prototypowanie pozwala zapobiec błędnej interpretacji wymagań systemu, co jest bardzo częstym zjawiskiem, gdyż mogą powstać rozbieżności pomiędzy językiem stosowanym u zleceniodawcy zastosowanym przy specyfikacji wymagań a językiem stosowanym przez informatyka odczytującego intencje zleceniodawcy. Bardzo często prototypowanie dotyczy warstwy prezentacji – czyli interfejsu użytkownika.



Rysunek 5. Schemat modelu prototypowania

Dzięki zastosowaniu prototypu można uniknąć wzrostu kosztów poprzez zminimalizowanie ryzyka błędnego postrzegania wymagań, co może być bardzo prawdopodobnym efektem w przypadku podejścia według modelu kaskadowego.

Należy zwrócić uwagę, że atrapa (prototyp) jest prawie "pusty" a więc jest łatwy do zmiany. Jednocześnie poddanie go krytycznej rewizji i akceptacji przyczynia się do zwiększenia zrozumienia całego systemu przez programistów co do rzeczywistych (funkcjonalnych) do potrzeb klienta. Ponadto atrapa pozwala odbiorcy już na wstępnym etapie roz-

woju zobaczyć w przybliżeniu jak będzie wyglądało docelowe rozwiązanie (ang.: look-and-feel). W efekcie mamy sytuację, w której wszystkie wymienione czynniki przyczyniają się do redukcji kosztów związanych z budową nowego oprogramowania.

Przy budowie prototypów możemy wyróżnić następujące podejścia:

- "pusty" prototyp dla szybkiego sprawdzenia koncepcji systemu
- użycie szablonów w celu szybszego stworzenia interfejsów
- rozrysowanie ekranów na arkuszach papieru, w PPT, itp.
- implementacja wybranych modułów programistycznych, np. bez ostatecznych wydruków, itp.
- implementacja kodu działającego dla większości funkcjonalności
- implementacja kodu dla wybranych danych testowych w celu sprawdzenia koncepcji i poprawności kluczowych funkcjonalności oprogramowania

Przy zastosowaniu tego modelu należy zawsze przypominać odbiorcy, że testuje on atrapę a nie "prawie" gotowy produkt. Należy go uświadomić, że od atrapy do pełnej realizacji jest ciągle wymagany pełny cykl rozwoju oprogramowania, patrz rysunek 5, co pozwoli to na uniknięcie nieudomówień co do jego rzeczywistego stanu zaawansowania w budowie oprogramowania.

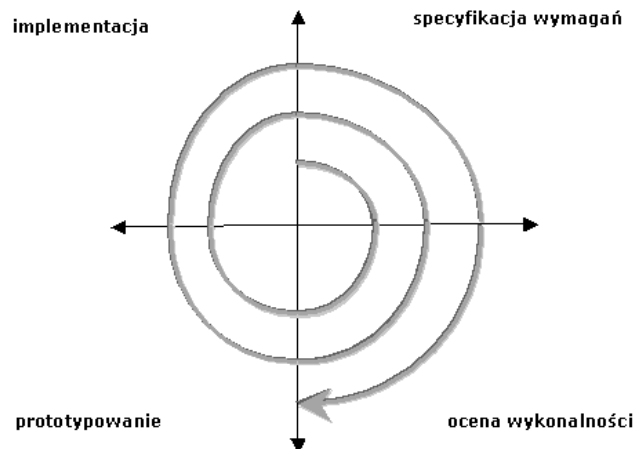
2.5 Model: spiralny

W tym modelu, proces tworzenia oprogramowania ma postać spirali (ang.: spiral model), której każdy zwoj reprezentuje różne fazy procesu. Rozmiar zwoju, a więc jego odległość od środka spirali, reprezentuje rozmiar tworzonego oprogramowania lub, inaczej mówiąc, rozmiar jego funkcjonalności. Najbardziej wewnętrzna pętla przedstawia początkowe stadium projektu. Odległość między zwojami pętli może odzwierciedlać zasoby, dzięki temu możemy pokazać jakie jest wykorzystanie zasobów

na danym etapie projektowania i jak się ma wykorzystanie zasobów do rozwoju funkcjonalności.

W modelu spiralnym każdy zwój przebiega przez cztery sektory, w których można zastosować dalszy podział na etapy składowe kolejnego cyklu:

- **Specyfikacja Wymagań** – w tej części cyklu dokonuje się definicji konkretnych celów wymaganych w tej fazie przedsięwzięcia, której odpowiada właśnie pojedynczy zwój spirali; dodatkowo podejmuje się oceny ograniczeń a następnie ustala się plany realizacji tego pojedynczego cyklu.
- **Ocena wykonalności** – na tym etapie dochodzi do rozpoznawania i usuwania potencjalnych zagrożeń poprzez przeprowadzenie szczegółowej analizy zagrożeń, ich źródeł a następnie sposobów ich zapobiegania.
- **Prototypowanie i zatwierdzanie** – tworzenie wersji testowych oprogramowania w oparciu o przyjęty w poprzedniej fazie projekt.
- **Implementacja i weryfikacja** – w tej części następuje wytwarzanie produktu końcowego oraz dokonywana jest recenzja postępu prac i planowanie przejścia w kolejną fazę (na schemacie - zwój) projektu bądź jego zakończenie.



Rysunek 6. Schemat modelu spiralnego rozwoju oprogramowania

Widoczną tu cechą modelu spiralnego jest wyróżnienie jako pierwszego - etapu oceny wykonalności a więc potraktowanie zagrożeń realizacji projektu jako szczególnie istotne. Dobrze zidentyfikowane zagrożenia i przygotowanie projektu, w celu im zapobiegania lub minimalizacji, dają w efekcie oprogramowanie o wysokiej niezawodności.

Okazuje się również, że model spiralny sprawdza się także w sytuacjach, gdy wymagania użytkownika ewoluują, a więc dziedzina projektu nie jest tak stabilna. Poszczególne zwoje spirali mogą być opracowywane w oparciu o nowe wymagania. Zwykle, w takich przypadkach, każdy nowy cykl wymaga podejmowania decyzji o kontynuacji i dalszym rozwoju projektu.

Stosując się do modelu spiralnego możliwe jest budowanie niezawodnego oprogramowania ponieważ, ze względu na istnienie fazy oceny ryzyka możliwe jest uniknięcie lub wczesne wykrycie błędów, co ma pozytywny wpływ na końcowy efekt co do kosztów związanych z całym przedsięwzięciem. Dodatkowo, pozytywnym aspektem stosowania modelu spiralnego jest otwartość, to znaczy zawsze będziemy mieli możliwość dalszego rozwijania projektu - spirala pozostaje zawsze "otwarta". Niemniej jednak warto zwrócić uwagę, że tworzenie oprogramowania w oparciu o model spiralny wymaga doświadczenia w prowadzeniu tego typu przedsięwzięć.

2.6 Inne modele

Istnieje wiele modeli innych niż te, opisane w poprzednich podrozdziałach. Warto zwrócić uwagę, że dotychczas opisywane modele skupiały swoją uwagę na wymaganiach, jak model kaskadowy lub też na zasobach, jak model przyrostowy. Warto wspomnieć o nowo powstałych modelach, które koncentrują się bardziej na ludziach i ich osobowości a nie na narzędziach i procesach. Są to modele jeszcze nie w pełni rozwinięte i sformalizowane, przez co nie są one jeszcze stosowane z pełnym zaufaniem w przedsiębiorstwach, ale zdobywają sobie coraz większą popularność, a ich zastosowanie - jak się okazuje - w pewnych okolicznościach daje najlepsze rezultaty.

Na początku lat 2000 zaczęto rozwijać modelowanie oparte na zdyscyplinowanym zarządzaniu projektem, które zakłada bardzo częste inspekcje wymagań i ich rozwiązań, nazwane **programowaniem zwinnym** -

agile software engineering. Według postulatów jej twórców należy dążyć do wytwarzania kodu lekkiego i wystarczającego do zrealizowania funkcji. Wiedza, że doskonała komunikacja jest niemożliwa, uwalnia od konieczności próbowania osiągnięcia tej perfekcji. Zamiast tego należy uczyć się zarządzania niekompletnością komunikacji. Zamiast próbować uczynić dokumentację wymagań lub model projektowy w pełni wyczerpujący dla każdego, należy zatrzymać się w momencie, gdy dokument jest już wystarczający dla celów zaplanowanych przez odbiorców. Zarządzanie niekompletnością komunikacji jest kluczowe dla podejścia elastycznego tworzenia oprogramowania. (Cockburn, 2000)

Celem programowania zwinnego jest bardziej działający kod - a mniej doskonała dokumentacja. Ponadto programowanie zwinne skupia się głównie na adaptacji zmian wymagań, niż na realizacji zdefiniowanego wcześniej planu. Manifest tej metody zaleca stosowanie się do następujących postulatów:

- na każdym etapie rozwoju wymagana jest prostota, gdzie punkt ciężkości przeniesiony jest na aspekty techniczne oraz dobry projekt,
- osiągnięcie satysfakcji odbiorcy poprzez możliwie najkrótszy czas dostarczenia oprogramowania – nie bacząc na formę produktu,
- działające oprogramowanie jest dostarczane często raczej w wymiarze czasu mierzonym tygodniowo niż miesięcznie czy kwartalnie,
- bliska, codzienna współpraca pomiędzy odbiorcą i uczestnikami projektu,
- bezpośredni, również nieformalny, kontakt jako najlepsza forma komunikacji w zespole wszystkich osób zaangażowanych w projekt,
- podstawową miarą postępu jest działające oprogramowanie,
- późne zmiany w specyfikacji nie mogą mieć destrukcyjnego charakteru na zrealizowany już proces wytwarzania oprogramowania,
- samowystarczalność zespołów oraz regularna adaptacja do zmieniających się wymagań.

Wszystkie postulaty tej metody nastawione są jest na bezpośrednią komunikację pomiędzy członkami zespołu, minimalizując i nieco bagatelizując potrzebę tworzenia formalnej dokumentacji. Jeśli członkowie zespołu są rozproszeni to do codziennej, niezbędnej w tym podejściu komunikacji stosuje się nowoczesne narzędzia od tych najbardziej rozpowszechnionych, jak e-mail, chat na wideo-konferencjach sieciowych skończywszy.

Kolejną, wartą zaznaczenia metodą modelowania projektu, która zyskuje na popularności jest **programowanie ekstremalne** (ang.: extreme programming), w skrócie *XP*. Jest to nowy, „lekki” nurt w tworzeniu oprogramowania, w którym wykorzystuje się zasady minimalizacji, prostoty, komunikacji, wymiany zdań oraz oportunistu w dobrym tego słowa znaczeniu. Programowanie ekstremalne sprawdza się szczególnie w małych zespołach programistów i twórców aplikacji, powoływanych do szybkiego rozwoju oprogramowania w środowisku często zmieniających się i często pojawiających się nowych wymagań. W wielu firmach, szczególnie w tych szybko rozwijających się i o dużym potencjale intelektualnym, choć to brzmi nieco paradoksalnie, jedyną stałą rzeczą jest - zmiana. W takim środowisku praktyki programowania ekstremalnego sprawdzają się bardzo dobrze. *XP* zostało również zorientowane na uwzględnianie problemów ryzyka projektu. Ryzyko projektu zwiększa się, gdy dostarczenie oprogramowania jest wymagane w krótkim terminie albo gdy projekt jest nowym wyzwaniem ze względu na przykład na przyjęcie nowego języka programowania lub wybór nowej platformy implementacji itp. Postulaty programowania ekstremalnego mają na celu zmniejszenie tego ryzyka. Zasady programowania ekstremalnego wyrażają się w następujących założeniach (Beck, 1999):

- istnienie etapu planowania - umożliwia odbiorcy zdefiniowanie funkcjonalności oraz pożądaných mechanizmów, przy czym kryterium tego, co ma być zrobione są szacunki nakładów dostarczone przez programistów
- małe, ale częste uaktualnienia – twórcy wprowadzają wcześnie do produkcji proste systemy i aktualizują je często w krótkich wydaniach,
- słownik pojęć – twórcy i zleceniodawcy posługują się wspólnym systemem nazw i wspólnym opisem systemu, dzięki czemu ułatwiona jest komunikacja i weryfikacja tworzonego oprogramowania,

- czytelny projekt - program tworzony według zasad XP powinien być możliwie najprostszym programem spełniającym zadane wymagania,
- testowanie - twórcy cały czas koncentrują się na sprawdzaniu poprawności oprogramowania. Oznacza to, że w pierwszej kolejności opracowywane są testy, a następnie oprogramowanie spełniające wymagania zawarte w testach; rolą odbiorcy jest dostarczenie testów akceptacyjnych, co pozwala im mieć pewność, że zapewnione będą postawione przez nich wymagania,
- przepisywanie kodu (ang.: refactoring) - twórcy udoskonalają projekt systemu w czasie jego tworzenia poprzez utrzymywanie przejrzystego i poprawnego, oprogramowania, tj. bez redundancji kodu, dobrze opatrzonego w komentarze, prostego ale równocześnie kompletnego,
- programowanie w zespołach dwuosobowych – twórcy stosujący programowanie ekstremalne piszą cały kod produkcyjny w parach, dwóch programistów pracujących razem na jednej maszynie. Okazują się, że programowanie w parach daje lepsze oprogramowanie w podobnym lub nawet krótszym czasie, niż w wypadku tych samych programistów pracujących niezależnie,
- pełny dostęp do całego kodu - cały kod należy do wszystkich programistów. Pozwala to zespołowi szybko pracować, ponieważ w momencie wystąpienia potrzeby zmian, modyfikacja może zostać wykonana bezzwłocznie,
- ciągła integracja - system jest przebudowany i testowany wielokrotnie w ciągu dnia. To pozwala wszystkim twórcom być na bieżąco, umożliwiając tym samym szybkie postępy prac,
- kultura pracy - zmęczeni programiści popełniają więcej błędów. Zespoły XP nie pracują w nadgodzinach, mają również nielimitowany, elastyczny reżim prac zachowując tym sposobem świeżość, zdrowie i efektywność.
- ciągły kontakt z odbiorcą - projekt XP jest kierowany przez wyznaczoną osobę, która ma upoważnienie do określania wymagań, ustalania priorytetów i udzielania odpowiedzi na

pytania programistów co przyczynia się do poprawy komunikacji przy mniejszej objętościowo dokumentacji, która często jest najdroższą częścią projektu oprogramowania,

- utrzymanie spójnego standardu pisanie kodu - aby zespół mógł efektywnie pracować w parach i współdzielić własność kodu, wszyscy programiści muszą przestrzegać uzgodnionego standardu pisanie kodu oprogramowania.

Znajomość wszystkich metod, tych sformalizowanych, wymienionych w poprzednich paragrafach, jak również tych bardziej miękkich i lekkich, opisanych powyżej pozwala na wybór najbardziej odpowiedniego modelu w zależności od konkretnej sytuacji i warunków, w jakich przyjdzie prace rozwojowe nad oprogramowaniem.

3

Specyfikacja wymagań w procesie tworzenia oprogramowania

Wymagania opisują system informatyczny, jaki należy zbudować. Wymagania mogą dotyczyć funkcjonalności systemu, czyli tego, co system ma robić. Wymagania mogą mieć również charakter nie związany z funkcjonalnością samego systemu i mogą dotyczyć np. wydajności bądź bezpieczeństwa.

Specyfikacja wymagań jest procesem, który oparty jest na wzajemnej komunikacji pomiędzy wszystkimi stronami projektu. Dzięki temu można określić miejsce przyszłego oprogramowania w szerszym kontekście całej organizacji. Projektanci i twórcy aplikacji mają wówczas możliwość zrozumienia, na czym polega działalność biznesowa odbiorcy, w jaki sposób projektowane oprogramowanie ma realizować określone przez odbiorcę funkcje oraz z jakimi innymi systemami ma ono zostać zintegrowane.

Wymagania przybierają często postać specyfikacji funkcji, jakie system musi realizować. Przy definiowaniu wymagań określone są więc założenia funkcjonalne. Opisują one zadania, jakie system ma realizować. Definiują poszczególne wejścia i wyjścia systemu. Takie wymagania są często określane mianem wymagań behawioralnych.

Dodatkowo wymagania mogą dotyczyć warstwy prezentacji a więc tego, co użytkownicy systemu mogą zaobserwować na ekranach, jaki jest interfejs użytkownika, jak sformatowane są wydruki końcowe, itd. Często wymagania nie dotyczą bezpośrednio funkcjonalności czy wyglądu oprogramowania ale określają te atrybuty systemu które mają uwidocznić się w momencie wykonywania danego zadania, może to być określenie platformy, kompatybilności czy wydajności wytworzonego oprogramowania.

Zbiór wymagań stanowi kompletny opis oprogramowania i jest punktem wyjścia do dalszych etapów cyklu życia . Specyfikacja wymagań powinna być dokumentem, który cechuje się:

- poprawnością - każde założenie jest istotnie tym, co jest wymagane od systemu,
- weryfikowalnością - istnieje możliwość sprawdzenia, bądź zmierzenia tego, czy projektowany system spełnia założenia,
- jednoznacznością - każde stwierdzenie ma tylko jedną interpretację,
- kompletnością – specyfikacja zawiera wszystko, co ma znaczenie dla realizacji projektu,

- spójnością - brak jest wykluczających się założeń,
- zrozumiałością - forma dokumentu umożliwia różnym uczestnikom procesu tworzenia oprogramowania przyswojenie potrzebnych informacji,
- elastycznością - możliwe jest wprowadzanie zmian co do definicji wymagań,
- tożsamością - każde wymaganie istnieje z uwagi na istnienie uzasadnionej potrzeby w funkcjonalności.

Często założenia systemowe są trudne do opisania, co może negatywnie wpływać na jakość specyfikacji wymagań. Wynika to z braku dostatecznej komunikacji między odbiorcą a projektantem.

Okazuje się, że sposób zachowania systemu można opisać za pomocą przykładów użycia (ang.: use cases). Przypadek użycia to sposób interakcji jakiegoś zewnętrznego elementu z systemem. Reprezentuje on ciąg akcji, w którym podaje się, co system robi, ale nie definiuje sposobu wykonania. Na bazie przypadków użycia tworzy się odpowiednie diagramy które ułatwiają opis wymagań.

Przypadki użycia stosuje się, gdy użytkownik przeprowadza powiązaną funkcjonalnie sekwencję transakcji w ramach określonego dialogu z systemem. Umiejętność polega na identyfikacji celów użytkownika, a nie funkcji systemu. Jednym ze sposobów jest traktowanie zadania biznesowego użytkownika jako przypadek użycia i odpowiadanie na pytanie, w jaki sposób system komputerowy może je wspierać (Booch, Rumbaugh, & Jacobson, 2001). Diagramy przypadków użycia opisują zewnętrzny widok systemu i jego interakcję ze światem zewnętrznym. Świat zewnętrzny jest reprezentowany przez tzw. aktorów. Aktorzy reprezentują role osób lub inne systemów komputerowych wchodzących w interakcję z projektowanym systemem. Aktor to rola, nie osoba czy system. Jedna osoba może odgrywać różne role i jednocześnie ta sama rola może być odgrywana przez wiele osób. Sposób wykorzystania przypadków użycia opisano w rozdziale 4.5.

4

Zagadnienia analizy strukturalnej i projektowanie

Założmy, że chcemy wyprodukować samochód. Zależy nam na tym aby samochód był niezawodny i spełniał określone oczekiwania kierowcy. Takie przedsięwzięcie wymaga poważnego projektu, nie ma tu miejsca na improwizację czy nawet zwykły szkic. Projekt powinien zawierać nie tylko plany konstrukcyjne ramy czy też karoserii, ale również wszystkich układów wchodzących w skład pojazdu. Pełny projekt pozwala nam oszacować jakie materiały będą nam potrzebne do wyprodukowania pojazdu, jakie elementy, części i zespoły należy obliczyć, przetestować i zestroić, ile czasu zajmie nam wyprodukowanie poszczególnych zespołów a w konsekwencji całego pojazdu. W przypadku tworzenia oprogramowania sprawa jest równie złożona. Współczesne systemy informatyczne są bardzo rozbudowane, zarówno pod względem złożoności problemu, narzędzi i platform programistycznych wykorzystywanych do ich realizacji oraz konieczności zorganizowania pracy całego zespołu. Dodatkowym utrudnieniem może być brak wiedzy samego użytkownika jak właściwie ma wyglądać finalna wersja oprogramowania.

Dobre oprogramowanie powinno spełniać wszystkie szczegółowe wymagania użytkownika, jednocześnie też powinno cechować się niezawodnością, efektywnością, łatwością w jego konserwacji oraz dobrą ergonomią, ogólnie - powinno spełniać założone wymagania szczegółowe, o których mowa była w rozdziale 3. Stworzenie systemu nie polega na napisaniu kodu "na wagę". System powinien spełniać wymagania przyszłego użytkownika, co oznacza, że kluczową czynnością w całym procesie tworzenia oprogramowania jest faza początkowa, to znaczy jest poprawna analiza rzeczywistości a następnie właściwy projekt systemu.

4.1 Wprowadzenie

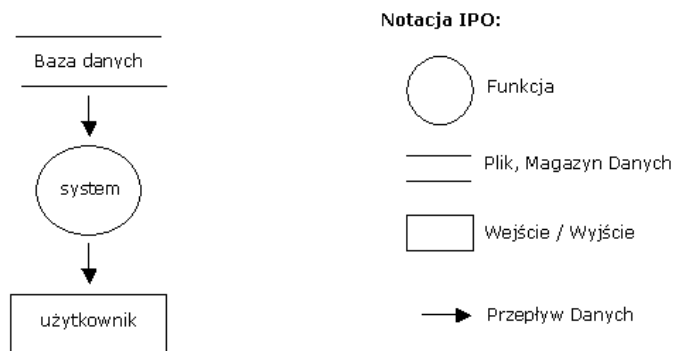
Projektowanie systemu realizowane jest za pomocą diagramów reprezentujących funkcje systemu przeznaczone do automatyzacji lub do przeniesienia ich do środowiska komputerowego, często też rozproszonego. Wykorzystywane w modelowaniu diagramy można scharakteryzować za pomocą trzech właściwości:

1. diagram modeluje fragment rzeczywistości,
2. różne diagramy modelują rzeczywistość z różnych perspektyw,
3. każdy diagram wykorzystuje zunifikowaną notację.

W inżynierii oprogramowania stosowane są trzy sposoby modelowania rzeczywistości: modelowanie zorientowane na procesy, na dane oraz modelowanie zorientowane na obiekty. W kolejnych punktach omówione zostaną poszczególne podejścia do opisu i analizy rzeczywistości.

4.2 Modelowanie zorientowane na procesy

W modelowaniu zorientowanym na procesy centralnym elementem jest, jak nazwa wskazuje - proces. Wszystkie dane wejściowe i wyjściowe są zgrupowane wokół danego procesu. Często diagramy te nazywane są diagramami IPO (ang.: Input-Process-Output). Natomiast technika modelowania używana w tym typie modelowania jest nazywana analizą (projektowaniem) strukturalną (ang.: SA - Structured Analysis).



Rysunek 7. Przykład diagramu IPO

W analizie strukturalnej każdy proces postrzegany jest jako na realizacja pewnego przekształcenia danych wejściowych w dane wyjściowe. Analiza strukturalna wykorzystuje również diagramy przepływu danych (ang.: DFD - Data Flow Diagrams). Diagramy te zawierają opis przepływu danych, magazynów danych oraz opis logiki procesów dla systemów i/lub podsystemów. Kolejnym etapem analizy strukturalnej jest opis podsystemów jako modułów programu za pomocą tzw. kart struktury (ang.: SC- Structure Cards). Wreszcie, logika programu jest opisywana w postaci tzw. pseudokodu.

W wielu przypadkach, szczególnie tych bardziej rozbudowanych systemów, modelowanie zorientowane na procesy, a więc dzielące system na zbiór niezależnych podsystemów, prowadzi do dużej fragmentacji. Powstają tak zwane wyspy informacyjne, które często okazują się trudne do wzajemnego połączenia a w efekcie może doprowadzić to do powstania niepotrzebnych interfejsów oraz nadmiarowych danych. Jest to zjawisko negatywne, które przyczynić się może do podwyższenia kosztów utrzymania oprogramowania. Integracja oprogramowanie jest oczywiście możliwa, jednak często jej koszty okazują się bardzo wysokie.

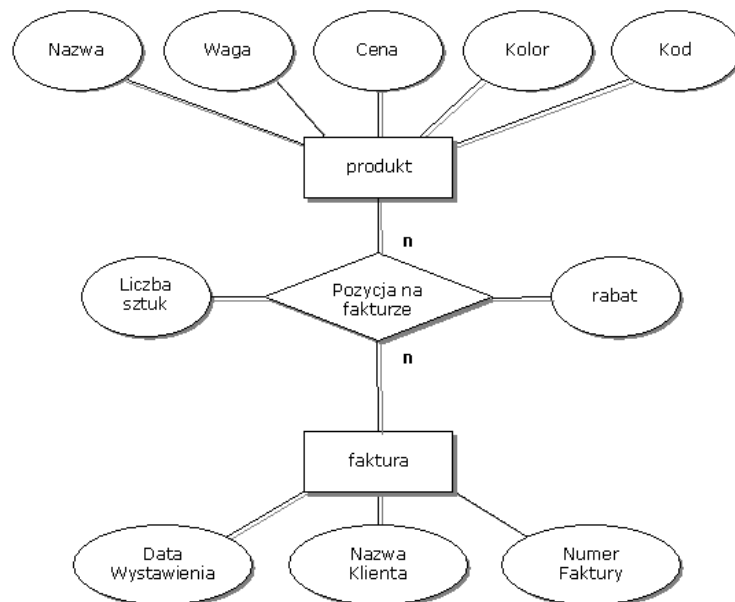
4.3 Modelowanie zorientowane na dane

Wraz z rozwojem technologii relacyjnych baz danych nastąpiło przeniesienie punktu ciężkości z procesów na dane, które w wielu przypadkach okazały się być centralnym i najbardziej stabilnym aspektem systemu informacyjnego - szczególnie w systemach wspierających operacje oparte na transakcjach. Zgodnie z postulatami bazy relacyjnej, dane umieszczone w tabelach są przez cały czas dostępne dla istniejącego jak i nowo tworzonego oprogramowania. Procesy biznesowe są zwykle zmienne i ewoluują wraz z rozwojem organizacji, restrukturyzacją lub zmianą metod zarządzania. W przeciwieństwie do tych procesów, dane są dużo bardziej stabilne i nie zmieniają się, jeśli tylko działalność firmy nie podlega fundamentalnym zmianom. Można więc uznać że osią systemu informatycznego są dane, wokół których budowane są odpowiednie procesy biznesowe. Modelowanie danych polega na opisywaniu tych rzeczy (zwanych w tej terminologii - encjami), o których dana firma chciałaby gromadzić informacje. Dodatkowo specyfikuje się wszystkie relacje pomiędzy nimi.

Już w latach 70-tych ubiegłego wieku został opracowany diagram modelujący dane, tzw. Diagram Związków Encji (ang.: ERD - Entity-Relationship Diagram). Rozwiązanie to jest powszechnie stosowane do dziś i jest podstawą tworzenia wszystkich aplikacji bazodanowych. W diagramie ERD encje są przedstawione w postaci prostokątów, związki są przedstawione jako romby na linii związków między encjami, a atrybuty są reprezentowane osobnymi kołami. Związki posiadają nazwy, które zwykle są kombinacją powiązanych encji i również mogą mieć pewne atrybuty. Na przestrzeni lat opracowano wiele pochodnych notacji,

wśród nich najczęściej stosowane są notacje Chena, ISO, Barkera (Oracle), IDEF1X oraz UML.

Rysunek 8 przedstawia przykładowy model dla prostego systemu fakturowania. **Faktura** jest zawsze wystawiona na jedną firmę zwaną **Klient**. Atrybutami faktury mogą być **Data Wystawienia**, **Nazwa Klienta** oraz **Numer Faktury**. Zamówione pozycje są zawarte w związku z encją produktu o nazwie **Pozycja na fakturze**. Faktura może zawierać więcej niż jeden produkt poprzez dodanie pozycji na fakturze. Z kolei pojedynczy produkt może występować na wielu fakturach. Po opracowaniu diagramu związków encji, postępując podobnie jak na powyższym przykładzie, można niemalże w sposób automatyczny utworzyć gotową strukturę dla relacyjnej bazy danych, składającą się z odpowiednich definicji tabel. Wówczas każda encja staje się tabelą z atrybutami jako kolumnami. Związki jednoznaczne, **1-n** bez atrybutów mogą zostać zaimplementowane jako tak zwane klucze obce. Związki z atrybutami lub związki wieloznaczne stają się w relacyjnej bazie danych oddzielnymi tabelami. W przykładzie, który ilustruje Rysunek 8 związek wieloznaczny **Pozycja na fakturze** odwzorowany będzie na oddzielną tabelę w relacyjnej strukturze danych.



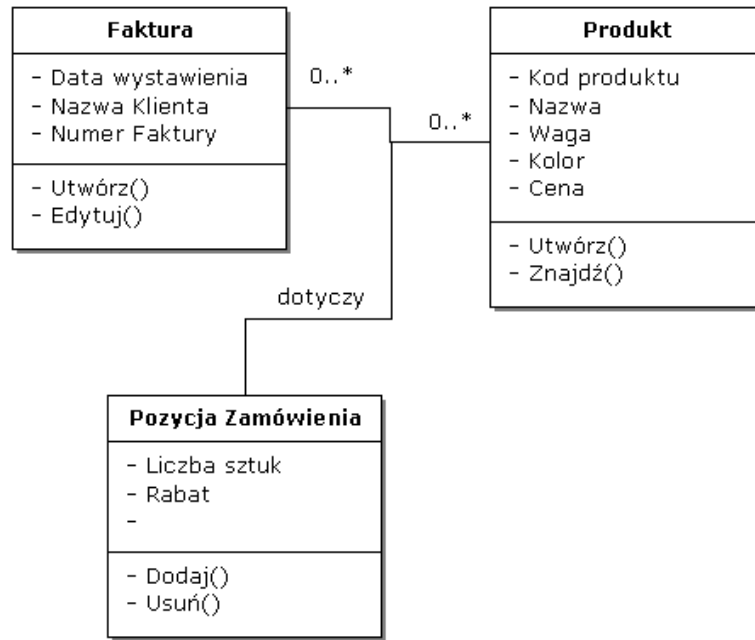
Rysunek 8. Przykładowy diagram związków encji dla faktury i pozycji na fakturze

4.4 Modelowanie obiektowe

W podejściu obiektowym każdy byt systemu jest widziany jako obiekt. Obiekt jest nie tylko strukturą danych. Obiekt posiada również szereg funkcji (zwanymi "metodami") umożliwiających zmianę jego stanu. Obiekty mogą komunikować się ze światem zewnętrznym za pomocą komunikatów. Prowadzi to do ukrywania informacji przed użytkownikami obiektów, co określane jest mianem hermetyzacji (ang.: encapsulation). Poprzez tzw. mechanizm dziedziczenia (ang.: inheritance) wspierana jest re-używalność czyli możliwość ponownego wykorzystania składników kodu.

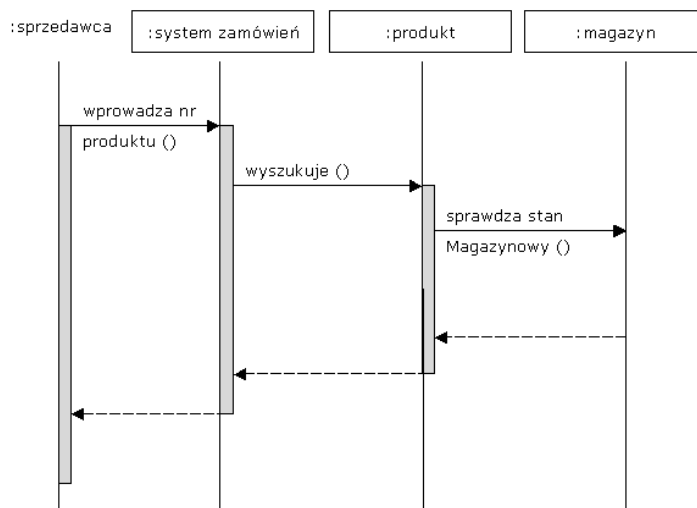
Na przestrzeni ostatnich lat opracowano szereg technik wspierających modelowanie obiektowe. W chwili obecnej standardem opisu stał się diagram klas języka UML -Unified Modeling Language. Jim Rumbaugh i Grady Booch zdefiniowali UML tak, aby uwzględnił wszystkie konstrukcje dostępne w każdej z metod, ale ze wspólną reprezentacją graficzną (Booch, Rumbaugh, & Jacobson, 2001). Z czasem język UML został wzbogacony o diagramy zaproponowane przez Ivara Jacobsona. Dzięki niemu została dodana składnia do definiowania wymagań za pomocą przypadków użycia. W tym miejscu zostaną zaprezentowane przykłady diagramów, który stosowane są najczęściej, ponieważ pozwalają one na niemal automatyczne generowanie kody aplikacji w dowolnym języku programowania zorientowanym obiektowo. Rysunek 9 przedstawia tzw. diagram klas, w którym wyspecyfikowane są klasy obiektów i związki pomiędzy nimi. Klasy definiują typy obiektów istniejących w systemie. Klasy posiadają atrybuty, które zwykle przechowują dane w prostej strukturze. Klasy zawierają również nazwy operacji (metod), które można przeprowadzić na danym obiekcie. Związki między poszczególnymi klasami pozwalają zamodelować połączenia pomiędzy obiektami i definiują na nich więzy określające liczbę instancji obiektów łączonych przez określony związek.

Jak można się zorientować, diagramy klas przedstawiają model systemu w sposób statyczny. Do modelowania dynamicznych aspektów systemu wykorzystuje się diagram sekwencji. Jest on, obok diagramu klas, najczęściej używanym diagramem w UML.



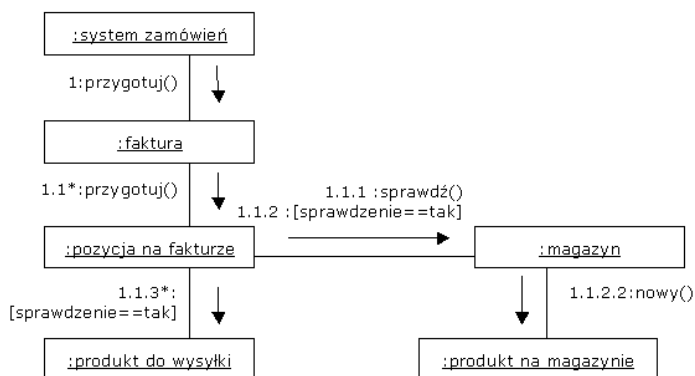
Rysunek 9. Przykład diagramu klas

Diagram sekwencji (ang.: Sequence Diagram) pozwala przedstawić interakcje, jakie mają miejsce zarówno między obiektami wchodzącymi w skład projektowanego systemu, jak i obiektami z otoczenia systemu. Diagramy sekwencji pozwalają zamodelować wymagania zdefiniowane przez przypadki użycia. Współdziałanie obiektów podporządkowane jest realizacji określonego zadania z uwypukleniem perspektywy czasu, a więc pozwala określić kolejność komunikatów w czasie. Rysunek 10 przedstawia przykładowy diagram sekwencji. Pionowe linie reprezentują etapy życia poszczególnych obiektów. Poziome strzałki reprezentują możliwe interakcje lub przesyłanie wiadomości pomiędzy obiektami biorącymi udział w omawianej sekwencji. Wiadomości mogą zawierać numery identyfikacyjne, nazwy operacji oraz ich parametry. Prostokąty na liniach życia obiektu reprezentują włączoną aktywność obiektu przy sekwencyjnych interakcjach inicjowaną wywołaniem transakcji. Transakcja pozostaje otwarta aż do zakończenia wszystkich sekwencyjnych operacji, które były przez nią wywołane.



Rysunek 10. Przykład diagramu sekwencji

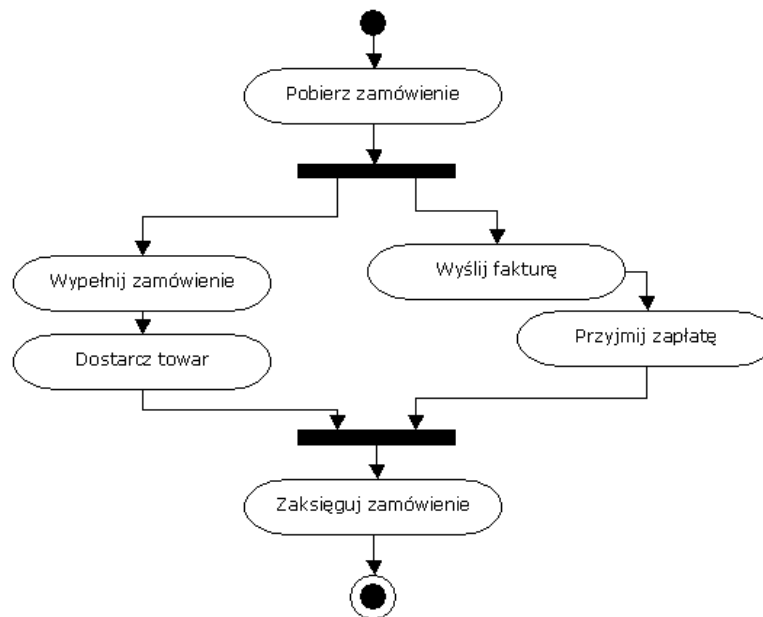
Diagramy współpracy (ang.: Collaboration Diagrams) odpowiadają funkcjonalnie diagramom sekwencji lecz są przydatne do modelowania sposobu współdziałania określonej grupy obiektów w celu realizacji danej funkcjonalności. Diagramu współpracy używa się po to, żeby zobrazować dynamikę systemu, a więc wzajemne oddziaływanie na siebie obiektów oraz komunikaty, jakie między sobą one przesyłają. Diagram współpracy może przedstawiać określone typy obiektów, określone operacje lub określone przypadki użycia. Rysunek 11 przedstawia przykład diagramu współpracy dla operacji przygotowywania faktury zamówienia.



Rysunek 11. Przykład diagramu współpracy

Diagramy stanów (ang.: State Diagrams) pokazują jak funkcjonalność obiektu zależy od jego stanu i jak zmienia się ten stan w wyniku otrzymywanych zdarzeń. Diagram stanów służy więc do tego, by pokazać w jakich stanach mogą być obiekty.

Diagramy aktywności (ang.: Activity Diagrams) mogą być używane do modelowania przepływu działań, chociaż zwykle są wykorzystywane do definiowania przepływu zdarzeń na poziomie funkcjonalnym, poza zakresem systemu. Diagram aktywności, zwany też diagramem czynności, służy do modelowania czynności i zakresu odpowiedzialności poszczególnych komponentów bądź użytkowników systemu. Diagram czynności jest podobny do diagramu stanu, jednak w odróżnieniu od niego, nie opisuje działań związanych z jednym obiektem a wieloma obiektami, pomiędzy którymi może występować komunikacja przy wykonywaniu różnych czynności. Rysunek 12 przedstawia przykład diagramu aktywności realizujący propozycję procesu realizacji zamówienia.



Rysunek 12. Przykład diagramu aktywności

Diagramy komponentów (ang. Component Diagrams) są używane do prezentacji struktury rzeczywistych składników oprogramowania użytych do budowy systemu. Mogą zawierać pliki źródłowe, kod współdzielony i programy wykonywalne. Prezentują typy składników systemu

i zależności między nimi włącznie z zależnościami pomiędzy interfejsami określonych komponentów.

Diagramy wdrożenia (ang. Deployment Diagrams) przedstawiają z kolei powiązania między oprogramowaniem a sprzętem. Są raczej stosowane przy modelowaniu dużych systemów. Diagram wdrożenia odzwierciedla fizyczną strukturę całego systemu, z uwzględnieniem zarówno warstwy oprogramowania jak i sprzętu. Jednostki oprogramowania są reprezentowane przez tzw. artefakty (czyli skompilowane wersje oprogramowania, które można uruchomić), dane oraz biblioteki. Stronę sprzętową reprezentują węzły, czyli poszczególne urządzenia komputerowe, komunikacyjne i przechowujące - powiązane ścieżkami komunikacyjnymi (np. połączeniem sieciowymi). Diagramy te są rzadko używane przy modelowaniu mniejszych i średnich systemów. Ponieważ diagramy wdrożenia posługują się zaledwie kilkoma symbolami, dlatego należy zwrócić uwagę na etykiety nadawane poszczególnym elementom. Pozwalają one doprecyzować znaczenie i funkcję każdej warstwy oprogramowania oraz warstwy sprzętu.

Na zakończenie tej pobieżnej prezentacji zbioru diagramów języka UML warto podkreślić istotną cechę tego języka. UML w pewnym sensie nie jest kompletny to znaczy nie uda się zamodelować przy jego pomocy każdego aspektu tworzonego systemu. Inaczej mówiąc brak jest formalnej zależności 1-do-1 pomiędzy diagramem a kodem realizującym ten diagram. Stąd też pojawiają się próby rozszerzenia UML o możliwości generowania bezpośrednio z niego gotowej aplikacji. Warto jednak pamiętać, że UML jest językiem modelowania. Natomiast model z założenia nie jest dokładną kopią rzeczywistego systemu, lecz takim jego obrazem, który pozwala poznać a także udokumentować jego najistotniejsze elementy.

Niezależna organizacja sprawująca pieczę nad rozwojem standardu UML, Object Management Group, zdefiniowała nową wersję UML, tzw. UML 2.0. Ta wersja UML opiera się na koncepcji znanej jako tzw. architektura sterowana modelem (ang.: MDA - Model Driven Architecture). Architektura ta propaguje podejście postulujące całkowite oddzielenie specyfikacji funkcjonalności systemu od jego platformy technologicznej, która może być użyta do jego implementacji. Tworząc aplikację opartą na MDA, należy najpierw zbudować model niezależny od platformy, tzw. model PIM (ang.: Platform Independent Model). Model ten powinien być następnie przekształcony na model specyficzny dla platformy docelowej, czyli tzw. model PSM (ang.: Platform Specific Model), taki jak na przykład J2EE, .Net lub CORBA. W specyfikacji UML 2.0 wykorzystuje również język ograniczeń obiektowych (ang.: OCL -

Object Constraint Language). Język ten służy do określania wszelkiego rodzaju ograniczeń, warunków początkowych i końcowych, itp. nakładanych na różne obiekty występujące w różnych modelach.

4.5 UML 2.0

UML 2.0 powstał na bazie wieloletnich doświadczeń analityków i projektantów. Jego zadaniem jest definiowanie, konstruowanie, obrazowanie i dokumentowanie części składowych dowolnego systemu komputerowego. Należy pamiętać, że jest to język modelowania, a więc opisuje, co system ma robić, a nie jak to zaimplementować. Analizując dowolne zadanie dotyczące inżynierii oprogramowania, możemy na nie popatrzeć z różnych punktów widzenia:

- sposób rozwiązania zadania (będącego przedmiotem systemu) - płaszczyzna projektowa
- funkcjonalność (co system ma robić) - płaszczyzna przypadków użycia
- procesy, jakie zachodzą w systemie - płaszczyzna procesowa
- sposób połączenia wszystkich części systemu - płaszczyzna implementacyjna
- sprzęt potrzebny do uruchomienia gotowego systemu - płaszczyzna wdrożeniowa

W poprzednich punktach przytoczono podstawowe typy poszczególnych diagramów, które łącznie używane są do opisu całej dziedziny problemu w sposób jak najbardziej kompletny. Poniżej przedstawione zostaną poszczególne elementy graficzne wchodzące używane do opisu system za pomocą diagramów UML.

Tabela 1. Elementy strukturalne - statyczne części modelu

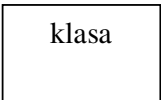


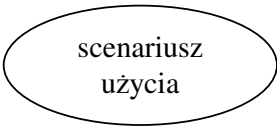
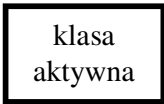
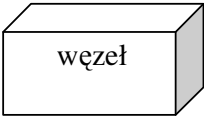
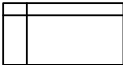
	Klasa: opis zbioru bytów, które mają takie same atrybuty, metody, związki oraz znaczenie
	Interfejs: zestaw operacji, które wyznaczają usługi oferowane przez klasę lub komponent
	Współpraca: interakcja, zestaw ról i bytów, współdziałających w celu wywołania pewnego zespołowego zachowania niemożliwego do osiągnięcia w pojedynkę
	Przypadek Użycia: opis zbioru ciągów akcji wykonywanych przez system w celu dostarczenia danemu aktorowi wyniku
	Klasa aktywna: zawiera obiekty, w skład których wchodzi co najmniej jeden proces lub wątek
	Węzeł: fizyczny składnik działającego systemu, reprezentujący zasoby obliczeniowe
	Komponent: fizycznie wymienna część systemu, która wykorzystuje i realizuje pewien zbiór interfejsów

Tabela 2. Elementy czynnościowe - dynamiczne części modelu


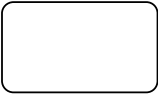
	Interakcja: wymiana komunikatów między obiektami w określonym otoczeniu i określonym celu
	Maszyna stanowa: ciąg stanów, jakie obiekt lub interakcja przyjmuje w odpowiedzi na zdarzenia zachodzące w czasie ich życia

Tabela 3. Elementy organizacyjne - grupujące i komentujące części modelu


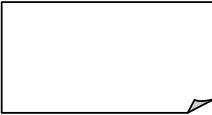
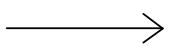
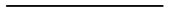
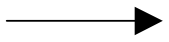
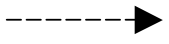
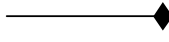
	Pakiet: służy do grupowania powiązanych elementów systemu. Jeden pakiet może zawierać podrzędne pakiety, diagramy lub pojedyncze elementy
	Notatka: symbol graficzny umożliwiający skojarzenie dodatkowych ograniczeń i objaśnień z pojedynczym bytem lub grupą bytów, nie ma wpływu na znaczenie modelu

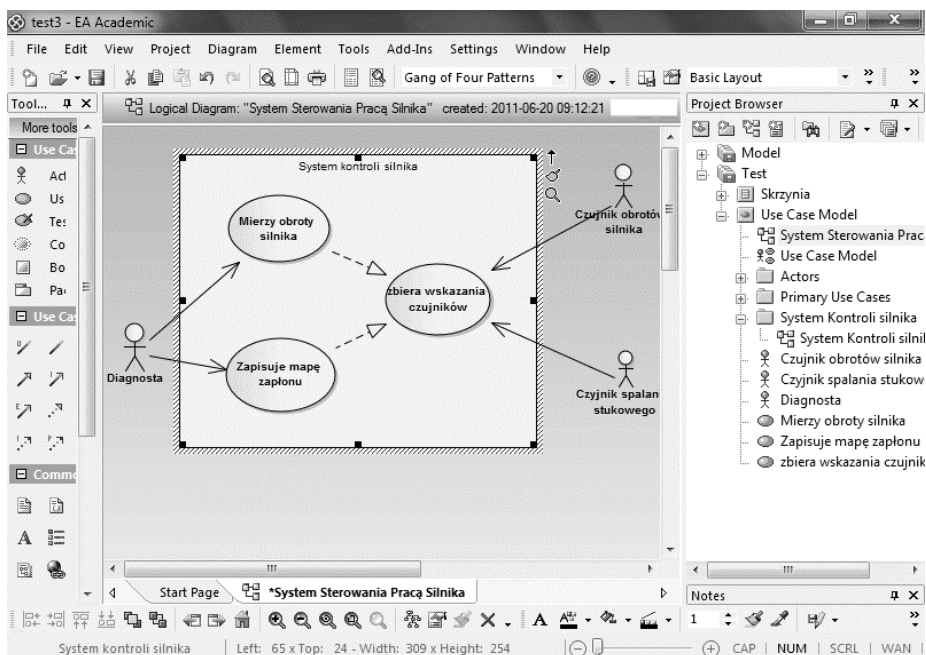
Tabela 4. Związki w diagramach UML

	Zależność: związek znaczeniowy między dwoma elementami, zmiany dokonywane w definicji jednego mogą mieć wpływ na znaczenie drugiego elementu
	Powiązanie: związek strukturalny, który określa zbiór relacji między określonymi obiektami
	Uogólnienie: związek między dwoma bytami: ogólnym (przodek) i szczegółowym (potomek)
	Węzeł: związek znaczeniowy między klasyfikatorami, z których jeden określa kontrakt, a drugi zapewnia wywiązanie się z niego
	Dodatki: role, np. pracownik - pracodawca liczebność: 1 jeden, 0..1 zero lub jeden, 0..* itd.. agregacja, agregacja całkowita

4.6 Generowanie modeli UML w narzędziach Enterprise Architect

Diagramy można rysować ręcznie na kartce papieru, choć przy bardziej złożonych modelach, i w zastosowaniach profesjonalnych, wykorzystuje się do tego celu wyspecjalizowane narzędzia. Na rynku dostępnych jest wiele programów oferujących pomoc w budowie diagramów UML, od

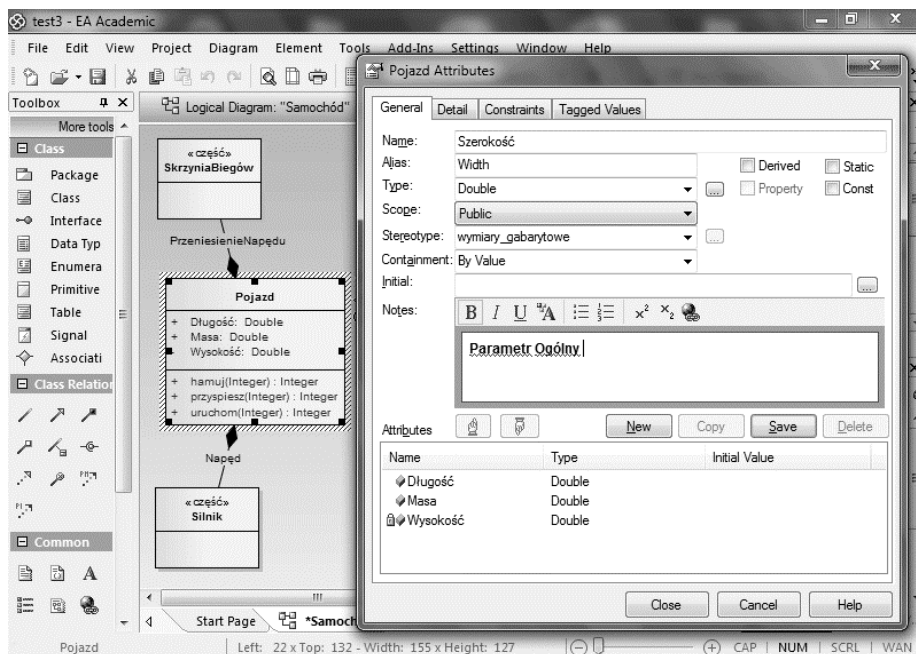
darmowych typu UMLet aż do licencjonowanych programów liderów tego typu oprogramowania firm Altova oraz Sparx Systems. W rozdziale tym opisano oprogramowanie firmy Sparx Systems o nazwie Enterprise Architect (EA). Narzędzie to w znaczny sposób ułatwia tworzenie diagramów a jednocześnie dostarcza szereg innych bardzo przydatnych w tym procesie funkcjonalności. Głównym elementem tego programu jest edytor diagramów, za pomocą którego tworzy się i modyfikuje dowolne diagramy, które mamy do dyspozycji zgodnie ze standardem UML. Program zapewnia również kontrolę składni, oraz hierarchiczną organizację struktury modelu za pomocą tworzenia drzewa pakietów.



Rysunek 13. Tworzenie diagramów w narzędziu EA

Jak pokazuje Rysunek 13, środowisko EA składa się z trzech paneli głównych i przypomina swym układem znane programistom środowisko Microsoft Visual Studio .Net. Po lewej stronie znajduje się przybornik narzędzi służących do wywoływania funkcji rysowania oraz edycji różnych elementów diagramów UML. Po prawej stronie znajduje się, stopniowo rozbudowywane, drzewo modelu, które może być dalej uważane za repozytorium, przechowujące całą hierarchię gotowych pakietów. Pakiety są to jednostki służące do grupowania modeli w pewne logicznie spójne całości. Są one reprezentowane są w drzewie modelu podobnie

jak foldery w Eksploratorze MS Windows. W poszczególnych pakietach umieszcza się różne modele UML. Dzięki możliwości stosowania pakietów możemy w jednym repozytorium przechowywać kilka różnych modeli, na przykład model silnika, model skrzyni biegów, itp. Jak wspomniano wcześniej, pakiety tworzą hierarchię, można je więc zagnieźdzać. Wewnątrz danego pakietu można tworzyć inne pakiety - choć w większości przypadków w pakietach umieszcza się już bezpośrednio konkretne elementy UML (klasy, interfejsy, przypadki użycia, itp.) a także diagramy (np. diagramy klas, diagramy sekwencji, itp.). W panelu centralnym, który zajmuje największy obszar środowiska EA tworzone są i edytowane konkretne diagramy. Poszczególne elementy diagramu tworzymy przez wywołanie odpowiednich narzędzi z przybornika po prawej stronie.



Rysunek 14. Zarządzanie modelem klas.
Definiowanie przykładowej klasy w EA

Rysunek 14 przedstawia sposób w jaki w narzędziu EA tworzy się nowy element poprzez wybranie odpowiedniego narzędzia z przybornika. W tym przypadku jest to klasa. Należy zwrócić uwagę, że nowoutworzony element pojawił się od razu zarówno na diagramie jak i w drzewie modelu. Możemy zmienić domyślną nazwę klasy (i jest to zalecane)

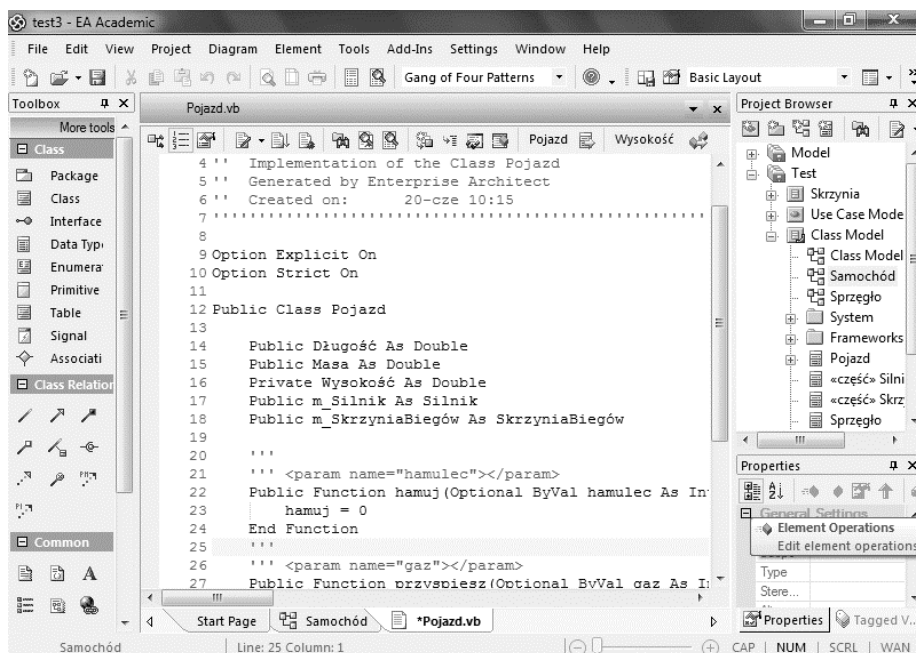
z Class1 na nazwę odpowiadającą naszemu modelowi oraz ustawić cały szereg innych parametrów definiujących ten element. Praca z tym narzędziem polega na budowie modelu UML poprzez dodawanie kolejnych elementów, które z kolei łączone są odpowiednimi relacjami (asocjacja, generalizacja, itd.) Program cały czas kontroluje i wymusza zgodność ze standardem UML. Należy zwrócić uwagę, że oprócz relacji pomiędzy elementami danego diagramu można również wprowadzać relacje i mapowanie pomiędzy różnymi diagramami. W ten sposób, poprzez wykorzystanie różnych typów diagramów, tworzony jest spójny model systemu, który przedstawia zarówno obiekty jak i interakcję pomiędzy nimi za pomocą odpowiednich diagramów UML.

4.7 Generowanie kodu źródłowego i inżynieria wsteczna

Kolejną ciekawą cechą narzędzia Enterprise Architect jest możliwość bezpośredniego generowania kodu źródłowego oraz re-interpretacji kodu wstecz czyli przeprowadzenia tzw. inżynierii odwrotnej (ang.: reverse engineering). Proces ten polega na tym, że zmiany dokonywane w kodzie źródłowym znajdują automatycznie swoje odbicie w modelu obiektowym, bezpośrednio w środowisku EA. Program ten potrafi generować kod w wielu językach programowania, przede wszystkim C++, Java, C# oraz Visual Basic .Net. Kod programu jest w przeważającej liczbie przypadków generowany na podstawie modelu klas. Powstaje wówczas tzw. szkielet kodu zawierający wszystkie klasy a także deklaracje ich atrybutów oraz metod. Same metody, czyli ich implementacja, muszą być oczywiście napisane przez programistę, choć w EA istnieją pewne możliwości wspomagania i tego procesu na podstawie zdefiniowanych odpowiednich modeli dynamicznych (np. diagramów sekwencji).

Rysunek 15 przedstawia możliwości Programu Enterprise Architect w zakresie automatycznego generowania oraz modyfikowania kodu źródłowego na podstawie odpowiedniego modelu UML. Dzięki takim funkcjom programu - droga od modelu, projektu aplikacji do utworzenia finalnego kodu źródłowego może ulec znacznemu skróceniu, przy jednocześnie poprawie jakości tego kodu, ponieważ posiadać on będzie lepszą

strukturę oraz może być o wiele lepiej zarządzany w sposób bardziej wizualny. Proces pisania kodu aplikacji zmienia się dzięki temu w proces wizualnego modelowania i tworzenia programu.

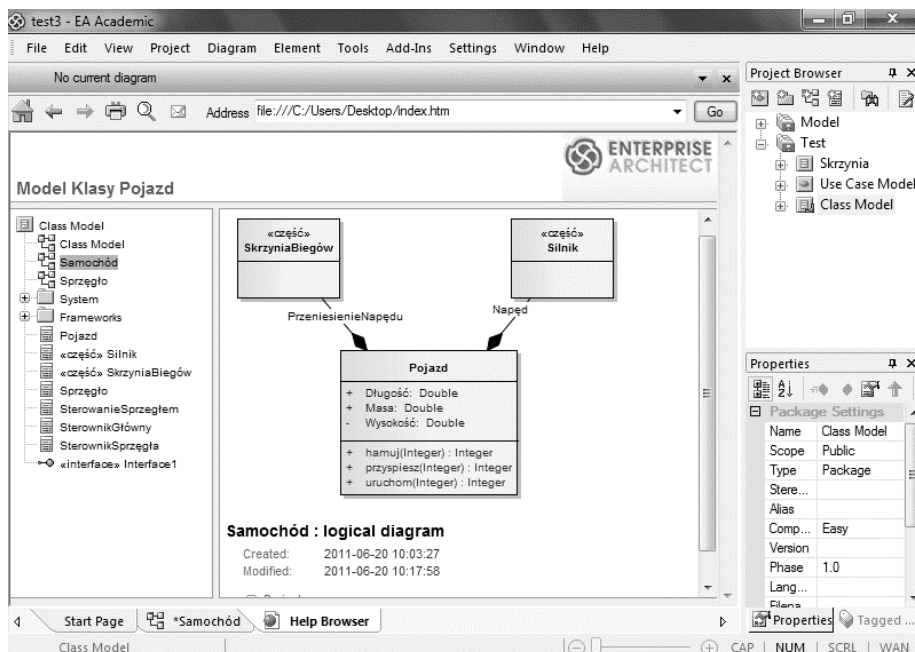


Rysunek 15. Szkielet kodu przykładowej klasy wygenerowany automatycznie za pomocą narzędzia EA

4.8 Generowanie Dokumentacji w Enterprise Architect

Jak wspomniano wcześniej, jakość oprogramowania, jakość tworzonego systemu informatycznego zależy od wzajemnego zrozumienia twórców tego oprogramowania z jego odbiorcami. Współpraca ta wymaga, aby obie strony miały pełne zrozumienie w zakresie tworzonego oprogramowania, każda ze swojej perspektywy. Z tego względu należy przewidzieć konieczność utworzenia pełnej dokumentacji tworzonego oprogramowania. Za pomocą programu Enterprise Architect możliwe jest tworzenie opisowych dokumentów tekstowych wraz z diagramami po-

chodzącymi z modelu aplikacji. Istnieje również możliwość wygenerowania dokumentacji w formatach umożliwiających publikację na stronach internetowych. Dostępny jest pełny tryb WYSIWYG. W programie tym dostępne są specjalne szablony dla wszystkich elementów modeli UML co pozwala na kompletne udokumentowanie ich właściwości oraz wszelkich związanych z nimi danych dodatkowych. Dostępne szablony można dalej rozwijać i personalizować stosując własne nagłówki, stopki, spisy treści, etykiety, grafiki, itd.



Rysunek 16. Dokumentacja modelu klas wygenerowana do postaci HTML za pomocą narzędzia EA

Dostępnych jest szereg opcji, które pozwalają generować dokumentację w sposób zautomatyzowany, z zastosowaniem różnych filtrów i kryteriów selekcji włącznie. Dostępny jest także dodatkowy generator raportów do formatu HTML, które mogą być przydatne np. dla kierowników projektów. Utworzone dokumenty mogą też stanowić właściwą dokumentację systemu, która może przybrać formę prezentacji bądź formalnego potwierdzenia kompletności projektu. Generowanie dokumentacji WWW może okazać się szczególnie przydatne, wówczas gdy system tworzony jest i użytkowany przez większą grupę osób, która dodatkowo może działać w środowisku rozproszonym. Rysunek 16 przedstawia fragment modelu oraz utworzoną na jego podstawie część dokumentacji

ROZDZIAŁ 4

html. Nie mniej jednak, największą wygodę w czytaniu modelu daje nie dokumentacja lecz przeglądanie modelu w samym programie Enterprise Architect.

5

Zarządzanie budową oprogramowania

Oprogramowanie inżynierskie jest zwykle bardzo złożonym przedsięwzięciem, w którym zwykle bierze udział wiele osób. Za pomocą modeli i narzędzi omówionych w rozdziale 4 istnieje możliwość zarządzania tworzeniem projektu oprogramowania. W tym rozdziale omówione zostaną zagadnienia dotyczące implementacji oprogramowania.

5.1 Wprowadzenie

Po procesie Analizy i projektowania systemu informatycznego kolejnym etapem jego rozwoju jest etap tworzenia oprogramowania. Na tym etapie używa się różnych języków programowania. Mówiąc najbardziej ogólnie, języki programowania to określenie pewnych sztucznych języków (w odróżnieniu do języków naturalnych, etnicznych), pozwalających na odpowiedni zapis zadania oraz sposobu jego wykonania przez komputer. Okazuje się, że wbrew powszechnemu przekonaniu, języki programowania są bardziej złożone od języków naturalnych. Sytuację komplikuje fakt, że najczęściej każdy typ procesora ma własny zestaw wewnętrznych instrukcji, co oznacza, że każdy program musi być ostatecznie przekształcony na konkretny procesor tak, aby był on zrozumiały dla danego typu maszyny, wyjątkiem może być tu język Java, choć i w tym przypadku pojawić się mogą pewne problemy implementacyjne dotyczące zgodności wersji i bibliotek pakietów.

Współcześnie oprogramowanie tworzone jest w tzw. językach wysokiego poziomu, których składnia i słowa kluczowe mają maksymalnie ułatwić rozumienie kodu programu dla człowieka, tym samym zwiększając poziom abstrakcji i ukrywając przed nim różnego rodzaju sprzętowe niuanse każdego typu procesora. Najczęściej, uruchamianie programów napisanych w językach wysokiego poziomu polega na użyciu dwu alternatywnych narzędzi: interpretera lub też kompilatora.

Kompilator jest specjalnym programem transformującym instrukcje napisane w danym języku programowania i przekształcającym je na niskopoziomowe instrukcje zrozumiałe bezpośrednio przez dany typ CPU. Przy takim podejściu, za pomocą określonego edytora, programista pisze kod programu w języku wysokiego poziomu - wiersz za wierszem. Powstały w ten sposób dokument zawiera tak zwane instrukcje źródłowe języka. W kolejnym kroku należy uruchomić program kompilatora języka i podać na wejściu utworzony wcześniej dokument zawierający instrukcje źródłowe, lub mówiąc w skrócie - źródło. W trakcie działania

tego kompilatora, w pierwszej kolejności analizowane są po kolei instrukcje źródła i sprawdzana jest jego poprawność składniowa; następnie, jeżeli jest ona w całości poprawna, tworzony jest w odpowiedni sposób dla danego CPU - kod wynikowy. Dzięki tej operacji istnieje pewność, że jedne instrukcje odnoszące się do innych w zadanym pliku źródłowym, są odpowiednio zdefiniowane w powstałym kodzie wynikowym. W wyniku działania kompilatora powstaje kod maszynowy, to znaczy taki, który dany procesor jest w stanie wykonać instrukcja po instrukcji. Kod ten musi jeszcze przejść fazę tzw. linkowania. W tej fazie powstały kod maszynowy jest konsolidowany z innymi, niezbędnymi do działania całego programu, bibliotekami funkcji. Te dodatkowe pliki to najczęściej biblioteki producenta kompilatora i inne wyspecjalizowane biblioteki, np. związane z obsługą wejścia i wyjścia. Proces linkowania zapewnia utworzenie wszystkich niezbędnych odwołań pomiędzy instrukcjami zawartymi w różnych plikach składowych. Dopiero taki kod można z powodzeniem uruchomić na danym komputerze. Utworzony jest plik który jest wykonywalny. Pliki wykonywalne dla platformy MS Windows mają standardowo rozszerzenie nazwy .exe (ang.: executable). W innych systemach nazwy plików wykonywalnych mogą być konstruowane w inny sposób, lub pliki takie mogą posiadać atrybut wykonywalności (systemy Unix).

Drugim sposobem wykonywania programów jest użycie Interpretera. Działa on w sposób odmienny. Interpreter analizuje i realizuje każdy wiersz kodu źródłowego (wiersz po wierszu) napisany w określonym języku programowania na bieżąco, w trakcie wykonania programu, to jest bez uprzedniej analizy i przygotowania wersji wykonywalnej. Jest więc to działanie ad hoc. Zaletą jest to, że program wykonywany jest niezwłocznie, bez procesu kompilacji i linkowania. Do wad tego rozwiązania zaliczyć należy szybkość wykonywania instrukcji, co przy skomplikowanych zadaniach obliczeniowych może mieć duże znaczenie. Z tego względu do zadań profesjonalnych nie stosuje się interpreterów. Kompilatory wymagają więcej czasu w momencie przygotowania postaci wykonywalnej programu, ale dzięki temu otrzymujemy kod który jest dobrze zoptymalizowany i działa bezpośrednio na wewnętrznych instrukcjach procesora, przez co jest niezwykle szybki, nieporównanie szybszy od kodu wykonywanego przez interpreter. Dodatkowo, postać wykonywalna programu ukrywa jego źródła, a więc kod źródłowy jest niewidoczny, niejako zaszyfrowany, dla użytkownika końcowego, co w pewnym stopniu zabezpiecza nasz kod przed zamierzonymi i niezamierzonymi modyfikacjami oraz pozwala zachować logikę programu wyłącznie dla twórcy co jest związane z zachowaniem własności intelek-

tualnej jego pracy twórczej. W przypadku kodu języków interpreto-
wanych kod programu pozostaje jawny.

Nawiązując do szybkości działania programów kompilowanych należy jednak pamiętać o tym, że działają one na wybranym typie procesora i konsekwencji również systemu operacyjnego. Jeżeli dany program miałby zostać uruchomiony na innej maszynie lub na innym systemie operacyjnym, to proces kompilacji należy przeprowadzić od początku za pomocą kompilatora dostępnego na inną, żadaną platformę. W rzeczywistości, ze względu na dostępność dość dużej liczby różnych typów procesorów i systemów operacyjnych, brak przenaszalności oprogramowania stał się istotnym ograniczeniem w rozwoju oprogramowania, chcąc stworzyć systemy działające na wszystkich typach komputerów działających w danej organizacji należy też zwielokrotnić wysiłki przez tworzenie i utrzymywanie oprogramowania w różnych postaciach wykonywalnych dla każdej z dostępnych platform. Od wielu lat próbowano temu zaradzić tworząc języki uniwersalne, przenaszalne z jednej platformy na drugą. Jakkolwiek koncepcja takiego kodu znana jest już od wielu dziesięcioleci, lecz języki oparte na tej koncepcji nigdy nie zdobyły popularności. Dopiero wprowadzenie języka Java i zawartej w nim możliwości kompilacji do tzw. kodu pośredniego pozwoliło w większości przypadków rozwiązać te dylematy - a sam język zyskał ogromną popularność tam, gdzie kwestie przenaszalności kodu były kluczowe, a więc w przedsiębiorstwach, a nawet w oprogramowaniu popularnych urządzeń użytku domowego sterowanych mikroprocesorowo. Testy wydajnościowe wykazują, że programy w Javie, szczególnie na współczesnych procesorach, praktycznie nie ustępują programom napisanym w innych językach kompilowanych na konkretną platformę.

W języku Java powstaje tak zwany kod pośredni, który może być uruchamiany na każdym komputerze, dla którego dostępna jest tzw. maszyna wirtualna Javy (ang.: JVM - Java Virtual Machine) będąca w dużym uproszczeniu rodzajem interpretera kodu pośredniego. Również firma Microsoft wprowadzając technologię .Net - skłania się w stronę budowy oprogramowania według zbliżonej koncepcji, wprowadzając swój język CLR (ang.: Common Language Runtime) będący pewnego rodzaju maszyną wirtualną wykonującą kod pośredni języka MSIL (ang.: Microsoft Intermediate Language). Oba rodzaje kodu pośredniego (kod bajtowy Java i MSIL) są tłumaczone do kodu maszynowego przez tzw. kompilator natychmiastowy (ang. JIT- Just In Time Compiler). Dla zapewnienia kompatybilności każdy język .Net musi być zgodny ze specyfikacją CIS (ang.: Common Language Specification) udostępnioną przez Microsoft. Obecnie zostały już wprowadzone wersje CLR działa-

jące na platformach innych niż Microsoft, tak więc można uruchamiać np. Visual Basic .Net na komputerach Linux lub mainframe, choć nie jest to jeszcze zjawisko powszechne.

5.2 Generacje języków programowania

Języki programowania można podzielić na różne generacje. Przy czym generacje nie odnoszą się tu do daty powstania danego języka, tzn. język należący do starszej generacji nie koniecznie musiał powstać wcześniej. Nawiązując do języków wymienionych uprzednio, Java należy do języków trzeciej generacji, a na przykład Visual Basic – piątej, pomimo tego, że Java powstała znacznie później niż Visual Basic.

Języki pierwszej generacji, oznaczane powszechnie skrótem 1GL, to języki maszynowe a raczej poziom instrukcji i danych, na których może bezpośrednio pracować procesor. W tym przypadku zarówno instrukcje, jak i dane są umieszczane w pamięci operacyjnej i nie istnieje rozróżnienie pomiędzy nimi.

Języki drugiej generacji, znane jako języki 2GL, są nazywane również assemblerami. Język taki zawiera te same instrukcje co kod maszynowy, ale instrukcje i zmienne mają już nazwy zamiast być jedynie ciągiem liczb. Program przekształcający instrukcje tekstowe do postaci maszynowej nazywany jest assemblerem. Termin "assembler" oznacza niskopoziomowy a jednocześnie czytelny dla człowieka sposób programowania poszczególnych procesorów. W praktyce istnieje wiele różnych dialektów tego języka, ponadto jest on zależny od danej architektury sprzętowej. Dlatego też istnieją różne odmiany tego języka dla każdego z rodzajów procesorów. Assembler jest generalnie sposobem komunikacji 1-do-1 z właściwym językiem maszynowym. Poniżej zaprezentowano kod źródłowy programu, którego zadaniem jest wyświetlenie na konsoli (na ekranie) klasycznego pozdrowienia 'Witaj!!!'. Jak łatwo zauważyć ten format jest dużo czytelniejszy niż rząd liczb, co miałoby miejsce w przypadku 1GL.

```

segment dane
    kom db "Witaj!!!",0Ah,0Dh,"$"

segment stos stack
    resb 64

segment kod
..start:
    mov ax, dane
    mov ds, ax
    mov ax, stos
    mov ss, ax

    mov dx, kom
    mov ah, 9
    int 0x21
    mov ax, 0x4C00
    int 0x21

end

```

Patrząc na kod tego programu trudno, szczególnie początkującym, znaleźć w nim logikę. W istocie, język asemblera jest jednym z nielicznych, które naprawdę potrafią przy pierwszym kontakcie wprowadzić w poruszenie początkującego programistę. Wynika to z tego, że język ten jest językiem niskiego poziomu to znaczy tłumaczy się go bezpośrednio do kodu maszynowego. Ze względu na tę właściwość nie jest to język, który nadaje się do nauczania programowania.

Języki trzeciej generacji, 3GL, to języki programowania wysokiego poziomu takie, jak na przykład C i Pascal. Przykład kodu źródłowego programu wykonującego tą samą funkcję co w poprzednim listingu:

```

#include <stdio.h>

int main(void) {
    printf("Witaj!!!\n");
}

```

Kompilator przekształca instrukcje języka programowania wysokiego poziomu na język maszynowy. W drugiej połowie lat dziewięćdziesiątych powstał język C++ który dawał duże możliwości programistyczne, choć wkrótce okazało się, że C++ był zbyt skomplikowany dla przeciętnego programisty a tworzenie oprogramowania w tym języku było dość żmudne. Większą popularność zdobył język Java. Testowy kod źród-

dłowy programu wypisującego pozdrowienie w C++ (standard ISO) miałby następującą postać:

```
#include <iostream>
int main()
{
    std::cout << "Witaj!!!" << std::endl;
}
```

W języku Java natomiast kod źródłowy takiego samego programu wyglądałby tak, jak przedstawiono to poniżej:

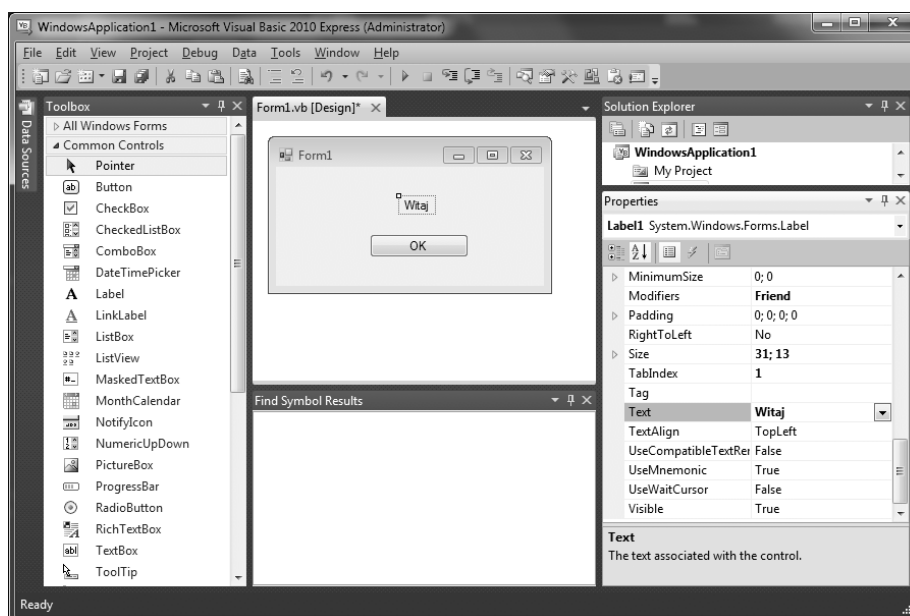
```
class HelloWorld {
    static public void main( String args[] ) {
        System.out.println( "Hello World!" );
    }
}
```

Dodatkową zaletą języka Java, o czym już wspomniano, jest jego przenaszalność. Polega ona na kompilacji kodu źródłowego do kodu pośredniego, który jest niezależny od procesora.

Języki czwartej generacji, 4GL są tak zaprojektowane, aby być jeszcze bardziej podobne do języka naturalnego. Języki te są zwykle wykorzystywane do dostępu do informacji w bazach danych. Najbardziej znanymi przedstawicielami tej generacji są SQL, Progress, Natural. Języki 4GL są językami deklaratywnymi - decyzyj o sposobie przechowywania i pobrania danych pozostawia się systemowi. W języku tym zapisuje się co ma zostać wykonane, a nie jak ma zostać to wykonane. Przykład, analogicznego do powyższych, pozdrowienia w SQL wyglądałby w sposób następujący:

```
SELECT 'Witaj!!!';
```

Języki następnej generacji, tzw. 4+, wykorzystują wizualny, graficzny interfejs do tworzenia programów źródłowych, które są zwykle kompilowane kompilatorami 3GL lub 4GL.



Rysunek 17. Przykład prostej aplikacji w środowisku Visual Basic Express 2010

Najbardziej znanymi przedstawicielem tej grupy języków jest środowisko Visual Basic. Rysunek 17 ilustruje program wyświetlający okno z pozdrowieniem ‘Witaj’ zrealizowany w aktualnej wersji środowiska Visual Basic Express 2010.

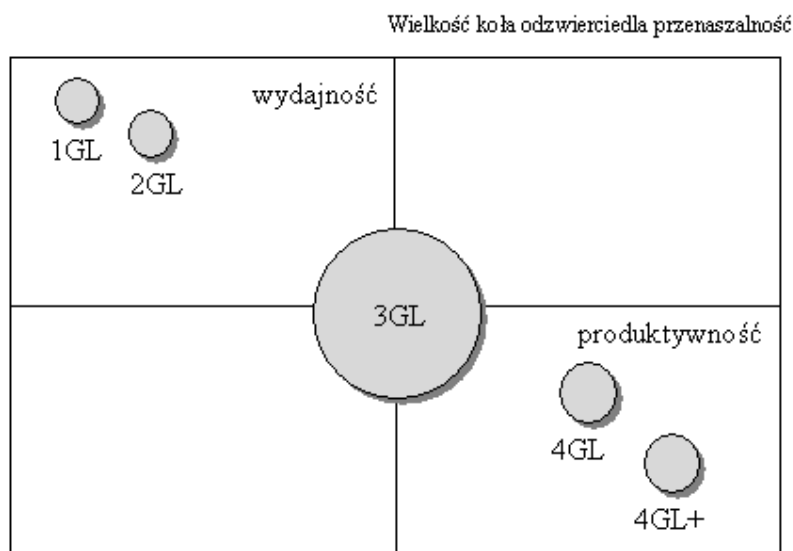
5.3 Cechy języków programowania

Trzema najważniejszymi cechami charakteryzującymi języki programowania są: wydajność, przenośność oraz produktywność. Czynniki te nie koniecznie związane są z konkretną generacją języka. Nawet w ramach jednej generacji można zaobserwować istotne różnice na przykład w produktywności. Według niezależnych badań przeprowadzonych przez niezależny instytut ISBSG (International Software Benchmarking Standards Group) okazuje się, że Java wykazuje najniższą produktywność ze wszystkich najbardziej popularnych języków. Dlaczego mimo tego jest

to jeden z najczęściej wykorzystywanych języków programowania? Do przyczyn tego stanu rzeczy zaliczyć możemy:

- Java jest wciąż językiem względnie nowym i jego środowisko programowania nie jest dojrzałe a doświadczenie w używaniu tej technologii wciąż się tworzy,
- narzędzia programistyczne dostępne dla języka Java jeszcze nie osiągnęły jeszcze takiego samego poziomu, co dla innych języków- nie są one jeszcze dostatecznie rozbudowane i nie posiadają tych wszystkich funkcji których oczekuje programista od tego typu narzędzi,
- zgodnie z naturą języka, wiele projektów w nim realizowanych to rozproszone aplikacje internetowe ze złożoną wielowarstwową architekturą, stąd większa złożoność a przez to słabsza produktywność,
- wiele małych i prostych projektów, które mogą być wykonane z bardzo wysoką produktywnością, są realizowane przy wykorzystaniu innych wciąż szerzej wykorzystywanych i znanych środowisk jak Visual Basic czy C,
- Wreszcie, z uwagi na względną nowość języka, realizowane w nim projekty należą przeważnie do projektów nowych, które z tego względu zawsze cechuje gorsza produktywność w porównaniu do wielu typowych projektów polegających na rozbudowie istniejących rozwiązań.

Nawet przy współcześnie stosowanym sprzęcie, charakteryzującym się bardzo wysoką wydajnością szybkość działania oprogramowania jest ciągle krytycznym czynnikiem dla wielu zastosowań. Przenaszalność, jako cecha nadawana oprogramowaniu, gdy może ono być używane w środowisku innym niż to, w którym zostało pierwotnie wygenerowane, bez konieczności wykonywania znacznych przeróbek. Oczywiście, systemy 4GL przystosowane do bardzo specyficznych środowisk będą pod tym względem gorsze niż inne. Rysunek 18 zawiera zestawienie różnych generacji języka. Analizując ten rysunek widać, że nie istnieje idealne rozwiązanie i wybór języka zawsze będzie jakimś kompromisem.



Rysunek 18. Zestawienie generacji języka programowania

5.4 Języki skryptowe

Języki skryptowe tworzą specjalną grupę języków programowania. Zanim zostaną one opisane należy określić czym jest skrypt. Według ogólnej przyjętej definicji, skrypt to program napisany w sposób czytelny dla człowieka; można poddawać go modyfikacjom; można uruchamiać go z różnymi parametrami wejściowymi; efektem jego uruchomienia jest wykonanie prostych operacji lub sterowanie wykonaniem innych programów. Język skryptowy jest z związku z tym takim językiem, który został zaprojektowany do pisania skryptów. Do grupy języków skryptowych zaliczyć możemy takie popularne języki jak Python, PHP, awk, sh, VB Script, Javascript itp.

Najwięcej języków skryptowych powstało w systemach Unix. W systemach tych skrypt jest plikiem tekstowym rozpoczynającym się wierszem podającym ścieżkę (specyfikację) interpretera. Kontynuując przykład powitania, skrypt mógłby wyglądać następująco:

```
#!/bin/bash  
echo 'Hello World'
```

Interpreter wiersza poleceń jest programem przyjmującym argumenty, ignoruje pierwszą linię i wykonuje odpowiednie działanie na zawartości skryptu, jak na przykład wypisanie tekstu. Ciekawym przykładem w systemie MS Windows jest Active-X Scripting Language - jest to obiekt COM zarejestrowany w systemie i implementującym odpowiedni interfejs do odbierania i wykonywania skryptów języka skryptów Microsoft VBScript. Z kolei VBScript jak i Javascript mogą być wykonywane przez przeglądarkę używaną do wspomagania wyświetlania stron internetowych. Języki skryptowe są to często języki interpretowane zaprojektowane z myślą o prostej interakcji z użytkownikiem. Niejednokrotnie polecenia używane w skryptach są bardzo skomplikowane, na przykład w języku sh większość poleceń to samodzielne programy. Języki te są często używane do jednorazowych zadań, najczęściej do zadań administracyjnych.

Języki skryptowe cechują się następującymi własnościami:

- kod źródłowy jest dostępny w czasie wykonania;
- nie są potrzebne fazy kompilacji i konsolidacji;
- zmienne, funkcje i metody nie wymagają deklaracji typów;
- typy mogą być składane w struktury heterogeniczne;
- istnieje ściśle określony sposób iteracji;
- interpreter może być osadzony w innej aplikacji;

5.5 Programowanie proceduralne i deklaratywne

Prawdopodobnie najbardziej znanym sposobem programowania jest programowanie proceduralne zwane często imperatywnym: programista określa w jawny sposób kolejność kroków, które należy wykonać dla

uzyskania wymaganego wyniku. W pojedynczym module bądź procedurze, który ma nadaną unikalną nazwę, zgrupowanych jest jeden lub wiele powiązanych ze sobą bloków instrukcji wykonujących w sposób kompletny zadaną funkcję. Jeśli ta sama funkcja jest potrzebna gdziekolwiek indziej w programie, instrukcja wywołania pozwala na odwołanie się do wcześniej zdefiniowanej. W ten sposób duże programy mogą być skonstruowane poprzez grupowanie procedur wykonujących dane zadania. Programy napisane w językach proceduralnych są krótsze i czytelniejsze, ale wymagają od programistów zaimplementowania każdej procedury w odpowiednio ogólny sposób, aby mogła być wykorzystana w różnych sytuacjach. Do grupy najczęściej używanych języków proceduralnych zaliczyć można na przykład języki Basic, Pascal i C.

W przeciwieństwie do języków proceduralnych, języki deklaratywne opisują relacje pomiędzy zmiennymi w obszarze funkcji lub reguł wnioskowania. Języki logiczne, na przykład Prolog, czyli języki programowania w logice, wykorzystują logikę matematyczną jako podstawę swojego działania. Program w języku logicznym składa się ze zbioru faktów i reguł warunkowych, które określają sposób wnioskowania zbioru faktów na podstawie innych faktów. Wiele programów należących do dziedziny sztucznej inteligencji (ang.: AI - Artificial Intelligence) jest napisanych w takich właśnie językach. Z kolei języki funkcyjne traktują procedury jak funkcje matematyczne i pozwalają przetwarzać w ten sam sposób co inne dane w tym programie, a funkcje należą do wartości podstawowych w tych językach. Przykładem języka funkcyjnego jest Standard ML (ang.: ML - Meta Language). Wiele innych języków, jak na przykład Lisp, ma podzbiór konstrukcji, które mają naturę czysto funkcyjną i często wykorzystuje rekurencję, chociaż ten język zawiera także elementy nie należące do tej kategorii.

5.6 Obiektowe języki programowania

Obiektowe języki programowania, czyli języki programowania zorientowanego na obiekty (ang.: OOP - Object-Oriented Programming), to języki, które są dalszym rozwinięciem języków funkcyjnych. W języku obiektowym programista definiuje nie tylko typy danych dla struktury danych, ale również typy metod, które mogą być do tej struktury danych zastosowane. W ten sposób struktura danych staje się obiektem, który

obejmuje zarówno dane, jak i skojarzone z nimi funkcje. Obiekty są pogrupowane w klasy, które z kolei są pogrupowane w hierarchie klas oraz pakiety.

Aby być w pełni obiektywnym, język programowania musi oferować przynajmniej trzy właściwości: hermetyzację, dziedziczenie i polimorfizm. Hermetyzacja, czyli ukrywanie kodu (ang. encapsulation), polega na tym, że każda klasa może ukryć dla innych klas z zewnątrz niektóre swoje elementy. Części kodu, które są wywoływane wyłącznie przez metody samej klasy są definiowane jako prywatne i są niewidoczne spoza klasy. Niektóre nowoczesne języki obiektywne, jak na przykład C++ i Java, mogą mieć nawet prywatne klasy wewnątrz klas czyli klasy prywatne dla klas zawierających.

Dziedziczenie jest metodą rozszerzania lub modyfikowania istniejących klas bazowych i tworzenia klas pochodnych odpowiednio powiązanych (pochodnych) z klasą bazową. Cokolwiek zostanie zmienione w klasie bazowej, to wszystkie klasy pochodne i pochodne od nich w kolejnych generacjach, automatycznie dostosowują się, dziedzicząc zmianę z klasy bazowej, a więc zmieniają swoje działanie. Jest to jeden z najważniejszych sposobów uzyskania dużej re-używalności kodu.

Pojęciem ściśle związanym z dziedziczeniem jest polimorfizm. Oznacza on, że obiekty różnych klas mogą być wykorzystywane w taki sam sposób, ale zależnie od klasy, będą się zachowywać odmiennie. Podczas pisania programu wygodnie jest traktować nawet różne dane w jednolity sposób. Na przykład, niezależnie czy należy wydrukować liczbę czy napis, czytelniej jest gdy operacja taka nazywa się w obu przypadkach tak samo, na przykład **drukuj**, a nie **drukuj_liczbę** i **drukuj_napis**. Jednak drukowanie ciągu znaków realizowane jest inaczej niż drukowanie liczby wobec czego istniały będą dwie implementacje polecenia drukuj. Polimorfizm pozwala stosować do tego celu wspólną nazwę, co daje wygodny, uniwersalny abstrakcyjny interfejs, niezależny od typu drukowanej wartości.

Pierwszym przedstawicielem wśród języków OOP był Smalltalk, opracowany na początku lat 70-tych ubiegłego wieku. Jednak Smalltalk nie znalazł ostatecznie szerokiego zastosowania. Z kolei jednym z najpopularniejszych języków obiektywnych pozostał C++, opracowany z w kolejnej dekadzie.

Dwie dekady po pojawieniu się Smalltalk firma Sun Microsystems wprowadziła język Java, który uzyskał ogólnoświatowe zainteresowanie i akceptację. Można powiedzieć, że w wielu aspektach Java reprezentuje

uproszczoną wersję C++, lecz dodaje także dodatkowe ułatwienia i dodaje możliwości przenaszalności, o czym wspomniano w poprzednim rozdziale. Dzięki temu w sposób szczególnie nadaj się do tworzenia interaktywnych aplikacji używanych w środowiskach rozproszonych.

Firma Microsoft nie mogła pozostać z boku, dlatego też, na przełomie wieków, został opracowany nowoczesny język C#. Co ciekawe # pochodzi od znaku chromatycznego podwyższającego ton w notacji muzycznej, przypomina również nachodzące na siebie znaki ++ - oba skojarzenia wydają się być tu uzasadnione. Język C# ma też wiele cech wspólnych z językami programowania Object Pascal, Delphi, C++ a także Java, dzięki czemu wielu programistów po prostu łatwo przesiada się na ten język.

5.7 Re - używalność kodu

Re-używalność oprogramowania oznacza użycie istniejących zasobów oprogramowania do spełnienia nowych wymagań powstałych w kontekście inżynierii oprogramowania. Kod programu jest jednym z przykładów zasobów oprogramowania, które używa się wielokrotnie. Inne dotyczą właśnie pisania samego kodu oprogramowania i obejmują odpowiednią znajomość dziedziny biznesowej, odpowiedniego sposobu projektowania i użycia odpowiedniej technologii co wymaga adekwatnych kompetencji programistów, specyfikacji wymagań. Zastosowanie re-użycia kodu ma dalszy wpływ na projekty, testy, architekturę, dokumentację, metody, procedury i narzędzia programistyczne. Wieloużywalność oferuje szereg korzyści, wśród których należy wymienić:

- zwiększoną produktywność,
- krótsze cykle tworzenia oprogramowania,
- wyższa jakość kodu,
- większa elastyczność,
- niższe koszty utrzymania oprogramowania.

Należy jednak zwrócić uwagę, że stosowanie re-użycia kodu może być kosztowne. Zapewnienie re-używalności wymaga dodatkowych nakładów związanych z koniecznością odpowiedniego szkolenia i możliwy

opór wśród personelu. Mogą także pojawić się trudności w znalezieniu i usystematyzowaniu zasobów, które mogą być zidentyfikowane i zakwalifikowane do wielokrotnego użycia, co zwiększa koszty utworzenia i utrzymania takich zasobów.

5.8 Aplikacje działające w środowiskach rozproszonych

W ciągu ostatnich lat obserwujemy istotny wzrost znaczenia i popularności rozproszonych aplikacji wielowarstwowych. Charakterystyczne jest dla nich rozdzielenie warstwy prezentacji (generowania strony interfejsu użytkownika) od warstwy przetwarzania danych, nazywanej również warstwą logiki aplikacyjnej. Zaletą jest znaczne uproszczenie administrowania takim systemem. Podział aplikacji na warstwy wymaga zastosowania tzw. komponentów. Komponent to wieloużywalny blok programowy, który może być wiązany z innymi komponentami w celu utworzenia kompletnego systemu. Komponenty mogą być umieszczane na różnych warstwach, na różnych serwerach w sieci i mogą komunikować się między sobą. Komponent funkcjonuje w pewnym kontekście nazywanym kontenerem. Przykładami kontenerów są strony WWW, przeglądarki, edytory tekstu itp.

Programowanie przy użyciu komponentów może być porównane do składania klocków Lego. Przykładem takiego modelu jest EJB (ang.: Enterprise JavaBeans) który jest serwerowym modelem programistycznym opracowanym w języku Java będący integralną częścią platformy J2EE - (ang.: Java 2 Enterprise Edition). EJB dostarcza pełnej funkcjonalności dotyczącej budowy komponentów i zarządzania nimi z poziomu języka Java. EJB wykorzystuje protokół komunikacyjny RMI (ang.: Remote Method Invocation). RMI jest niezależny od protokołów niższych warstw, więc może działać na jakichkolwiek protokołach, najczęściej jest to CORBA. Może on również działać bezpośrednio na warstwie TCP/IP protokołu sieciowego.

Analogiczną technologię wprowadził Microsoft w postaci ActiveX. Jest to zbiór obiektowych narzędzi oraz technologii programistycznych. Centralną jej część stanowi COM (ang.: Component Object Model). Jeśli

jest użyty w środowisku sieciowym, wraz z usługami katalogowymi i dodatkowymi mechanizmami, staje się modelem rozproszonym DCOM (ang.: Distributed Component Object Model). Komponenty w podejściu Microsoftu zwane są kontrolkami ActiveX.

6

Narzędzia wykorzystywane w inżynierii oprogramowania

W poprzednich rozdziałach dokonano opisu technologii informatycznych stosowanych do modelowania, projektowania a także implementacji oprogramowania w wybranych językach. W tym rozdziale omówiono zestaw narzędzi przydatnych bezpośrednio programistom w tworzeniu kodu oprogramowania.

6.1 Komputerowo wspomagana inżynieria oprogramowania CASE

Narzędzia komputerowo wspomaganej inżynierii oprogramowania CASE (ang.: Computer Aided Software Engineering) zapewniają zautomatyzowane wsparcie wielu metodom analizy systemowej i projektowania, najczęściej poprzez wsparcie dla języka modelowania UML. Są one szczególnie często wykorzystywane przez twórców aplikacji i dużych systemów informacyjnych. Narzędzia te dostarczają wyspecjalizowane środowiska, które automatyzują wiele czasochłonnych działań w ramach procesu tworzenia całego systemu. Wspomagają one takie żmudne zajęcia jak:

- rysowanie i edytowanie diagramów modelu,
- kontrola krzyżowa elementów pomiędzy różnymi częściami modelu,
- generowanie struktury kodu i schematów baz danych,
- generowanie dokumentacji systemu.

Narzędzia CASE możemy podzielić na dwie kategorie. Do pierwszej z nich zaliczymy narzędzia zorientowane na początkowe fazy cyklu tworzenia oprogramowania takie jak analiza wymagań, analiza systemowa, projektowanie. Są to tak zwane narzędzia CASE wysoko poziomowe (ang.: Upper CASE). Z kolei narzędzia niskopoziomowe (ang.: Lower CASE) zorientowane są bardziej na generowanie kodu, testowanie i analizowanie istniejących programów w ramach tak zwanej inżynierii odwrotnej. Wreszcie narzędzia CASE, obejmujące obie kategorie, są czasem nazywane zintegrowanymi lub iCASE. Interesującym mechanizmem oferowanym przez niektóre narzędzia CASE jest tzw. inżynieria

pełnego cyklu (ang. Round Trip Engineering). Model, a więc jego diagram, jest powiązany z fragmentem wygenerowanego kodu. Gdy dana część modelu jest tworzona lub modyfikowana, to powstaje odpowiedni kod źródłowy w określonym języku programowania. Z kolei przy zmianie kodu źródłowego przy użyciu edytora programisty, odpowiedni model jest automatycznie synchronizowany ze zmianami wprowadzonymi w kodzie. Funkcje te spełnia omówione w rozdziale 4.6 oprogramowanie Enterprise Architect. Kolejnym udogodnieniem mogą być narzędzia kontroli jakości, takie jak: określanie wartości odpowiednich miar, np. wiersze kodu (ang. LOC - Lines of Code) lub złożoność cykliczna używana do pomiaru stopnia skomplikowania programu, dalej - weryfikacja konwencji nazewnictwa lub weryfikacja standardów kodowania.

Narzędzia CASE oferują korzyści wynikające z używania wspólnego środowiska deweloperskiego przez wszystkie osoby pracujące nad danym projektem. Zmniejsza to wysiłek a więc i koszty związane z zarządzaniem, administracją, tworzeniem dokumentacji itp. Wszystkie dokumenty, źródła i referencje pozostają bowiem w centralnym systemie i są dostępne dla wszystkich zainteresowanych, nie zaś rozproszone wśród luźnych notatek na papierze, w szufladach biurków różnych programistów. Dzięki zaawansowanej krzyżowej kontroli spójności wzrasta także jakość samego modelu tworzonego systemu. Zastosowanie narzędzi CASE wpływa również na zmniejszenie czasu rozwoju systemu, a największemu skróceniu ulega czas pisania dokumentacji, ponieważ przy zastosowaniu CASE generowana jest ona całkowicie w sposób zautomatyzowany.

W latach 90-tych ubiegłego wieku narzędzia CASE były szeroko reklamowaną i modną nowinką. Jednak wielu programistów ostatecznie zniechęciło się do tych produktów ponieważ narzędzia te z reguły wymuszały jeden model rozwoju systemu. Poza tym, do efektywnego korzystania z tych narzędzi niezbędne było intensywne szkolenie personelu, trudne do zrealizowania przy dużej ówczesnie płynności kadr, co z kolei prowadziło do powiększania kosztów tego typu stanowisk pracy. Inne czynniki zupełnie niezwiązane z inżynierią oprogramowania, jak na przykład serie fuzji i przejęć powodowały, że inwestowanie w narzędzia CASE wiązały się z ryzykiem, które chciano odsunąć. Z powyższych względów narzędzia CASE, chociaż są bardzo dobrymi narzędziami, to jednak obecnie znajdują powodzenie jedynie w specyficznych zastosowaniach.

6.2 Zintegrowane środowiska deweloperskie

Zintegrowane środowiska deweloperskie (ang.: IDE - Integrated Development Environment) to specjalnie dedykowany rodzaj oprogramowania, w którym znajdują się odpowiednie narzędzia niezbędne do pracy z kodem źródłowym a więc narzędzia do edycji, kompilacji, konsolidacji, testowania, uruchamiania, i wykonania programów itp. przy czym są one odpowiednio zintegrowane i najczęściej wyposażone w różnego rodzaju komponenty ułatwiające wszystkie czynności a sposób wizualny.

Narzędzia IDE składają się z następujących funkcjonalności:

- **Zarządzanie projektem** - komponent który pozwala na zarządzanie wszystkimi bieżącymi projektami. Pliki należące do projektu mogą być na przykład rozmieszczane w strukturze katalogów systemu operacyjnego celem ułatwienia separacji plików i przeglądania/wyboru odpowiednich zbiorów.
- **Zarządzanie preferencjami** - możliwość wczytywania lub włączania plików ustawień w miarę potrzeby bez konieczności opuszczania środowiska programistycznego oraz narzędzia do porównywania różnych wersji plików źródłowych w czasie rzeczywistym.
- **Generator typowych aplikacji** - automatyczna generacja projektów i plików szablonowych dla określonych typów aplikacji, na przykład aplikacji typu okienkowego lub web-service,
- **Komponenty edycyjne** - edytory, najczęściej realizowane jako "wtyczki". Pozwala to na łatwe dodawanie nowych edytorów, takich jak edytory kodu, preferencji, ikon, edytory graficzne itd.,
- **Bieżąca Pomoc** - możliwość uzyskiwania pomocy kontekstowej, jak również przeglądania i wyszukiwania tematów związanych z oprogramowaniem,

- **Zintegrowane kompilatory** - zdolność budowania oprogramowania bezpośrednio ze środowiska IDE zaopatrzonego w różnorodne narzędzia weryfikacji kodu, kompilacji, budowy i przygotowania instalatora programu,
- **Zintegrowany debugger** - możliwość uruchamiania programów przy użyciu programu debugera bezpośrednio ze środowiska IDE,
- **Biblioteki komponentów** - możliwość wykorzystywania plików włączanych przez operacje dołączania oraz bibliotek dostępnych w określonych lokalizacjach,
- **Generatory dokumentacji** - ułatwienia do automatycznej generacji dokumentacji lub co najmniej mechanizmy wzajemnych odwołań pomiędzy poszczególnymi plikami, klasami, zmiennymi itd.

Warto zauważyć, że środowiska IDE, w przeciwieństwie do narzędzi CASE, koncentrują się na bardziej technicznych, narzędziowych aspektach tworzenia oprogramowania, natomiast samo modelowanie jest zwykle elementem drugorzędnym.

Zasadniczo, wykorzystywanie środowisk IDE przynosi znaczny wzrost produktywności. Ale należy pamiętać, że stosując dane środowisko IDE projekt zostaje do niego przywiązany i zależny, co może doprowadzić do niebezpieczeństwa, że w przyszłości będzie trzeba także opierać się na danym środowisku programistycznym. Należy więc mieć pewność, że będzie ono dalej rozwijane i wspierane.

Do najbardziej popularnych środowisk programistycznych IDE zaliczyć możemy:

- środowisko **Microsoft Visual Studio .Net** - popularny na systemach rodziny Windows, w którym można pisać aplikacje w różnych językach, przede wszystkim Visual Basic, C# oraz Java)
- dedykowane narzędzia programistyczne firmy **Borland**
- **Eclipse i NetBeans** - domyślnie stworzone dla Javy, które dodatkowo posiadają możliwość rozszerzania, w celu obsługi innych języków



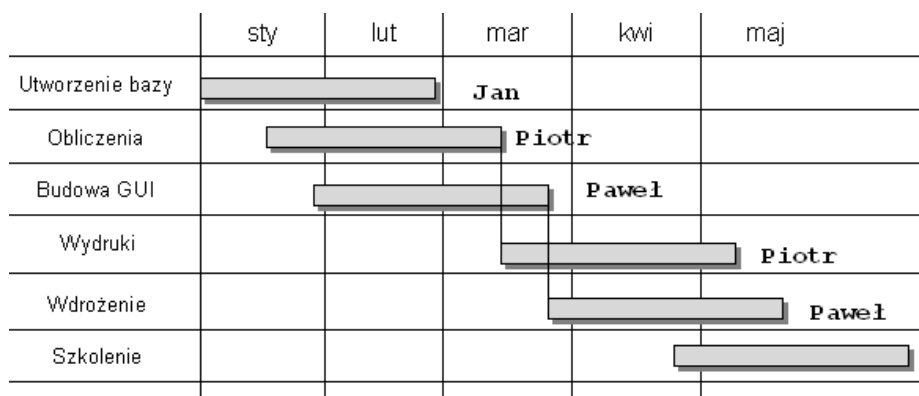
7

Kontrola i monitorowanie w inżynierii oprogramowania

Sukces projektu zależy od wielu czynników, o których większość omówiona była w poprzednich rozdziałach. Jednym z głównych czynników, który omówiony zostanie tutaj, jest planowanie samego przebiegu jak i zarządzanie przebiegiem realizacji projektu informatycznego. Identyfikacja i rozłożenie pracy na czynniki pierwsze stanowi punkt wyjście do całego planowania. Celem tego etapu jest podział pracy na możliwie małe, konkretne elementy, nazywane zadaniami. Po zidentyfikowaniu wszystkich zadań niezbędnych do realizacji projektu można przystąpić do budowy planu pracy. Daje to możliwość oszacowania czasu trwania i następstwa poszczególnych zadań, przygotowania odpowiednich zasobów ludzkich i finansowych, podziału obowiązków i innych aspektów organizacyjnych związanych z realizacją projektu.

7.1 Diagram Gantta

Henry Gantt opracował jedno z pierwszych narzędzi do zarządzania projektami, znane jest ono dziś jako diagramy Gantta. Wykresy tego typu to graficzna ilustracja działań koniecznych do realizacji danego przedsięwzięcia na osi czasu przy użyciu linii rozpoczynających się i kończących w zadanych punktach czasu. Diagramy Gantta uwidaczniają kamienie milowe i ułatwiają w ten sposób zorientować się czy projekt jest zgodny z harmonogramem.



Rysunek 19. Przykładowy diagram Gantta

Rysunek 19 przedstawia przykładowy diagram Gantta z planem projektu zdefiniowanym przez pewnego menedżera projektu. Jako że ta sam

osoba (zasób) ma wykonać dwa zadania w określonej kolejności, zakończenie funkcji Obliczenia jest traktowane jako kamień milowy. Z drugiej strony, zadanie Budowa_GUI również powinno być zakończone o czasie, gdyż determinuje rozpoczęcie zadania Wdrożenie zatem jest również kamieniem milowym. Oczywiście, koniec całego projektu jest automatycznie ostatnim kamieniem milowym.

Chociaż zaletą diagramów Gantta jest ich prostota, to jednak nie można na nich pokazać wielu, istotnych z punktu widzenia zarządzania projektem, rzeczy. W rzeczywistości, wzajemne relacje (pozasobowe) pomiędzy różnymi zadaniami nie są określone jawnie, tak samo jak związki pomiędzy zasobami i czasem potrzebnym do zakończenia zadania. Diagramy Gantta mogą pokazywać, że projekt jest poza harmonogramem, jednakże często nie dają informacji o tym, jak ponownie uzyskać zgodność z planem.

7.2 Harmonogramowanie PERT/CPM

Najpopularniejszą metodą dokumentowania relacji między zadaniami i określeniem ich względnego wpływu na projekt jest metoda PERT/CPM (ang.: Project Evaluation and Review Technique/Critical Path Method), Metoda ta powstała pod koniec lat 50-tych ubiegłego wieku, w celu ułatwienia planowania i harmonogramowania projektu Polaris prowadzonego przez Marynarkę Wojenną USA. W tym samym czasie firmy DuPont oraz Remington Rand wykorzystywali metodę CPM (będącą pewnym uproszczeniem metody PERT). Obie metody są podobne i wykorzystują model tak zwanej ścieżki krytycznej jako podstawę modelowania rzeczywistości. Metoda PERT kładzie większy nacisk na zarządzanie czasem, zaś w metodzie CPM środek ciężkości zwrócony jest w stronę problemów związanych z kosztami (zasobami), choć czas i koszt realizacji projektu są nie mniej istotne.

Sieci PERT/CPM dają bardziej szczegółowe spojrzenie na projekt. Składają się z działań i zdarzeń. Działania to operacje w projekcie konsumujące zasoby i zabierające czas; zdarzenia występują w określonym punkcie w czasie i reprezentują początek, koniec (lub oba parametry) działania. Tabela 5 przedstawia zależności między różnymi zadaniami przykładowego projektu.

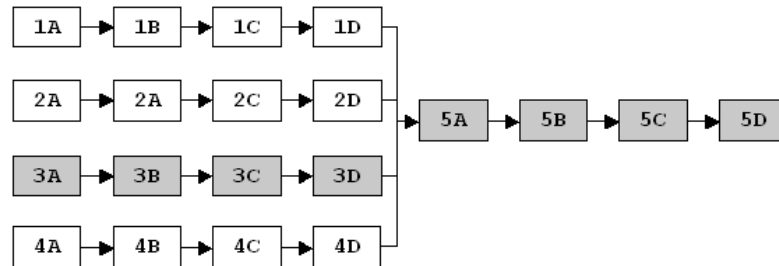
Tabela 5. Sieć PERT dla przykładowego zadania harmonogramowania

Funkcja	Zadania	Osoby	Czas [dni]	Zależności
1 Baza danych	a. Wymagania	Jan	2	2A;3A;4A
	b. Projekt		8	1A
	c. Kodowanie		8	1B
	d. Testowanie		8	1D
2 Obliczenia	a. Wymagania	Piotr	4	-
	b. Projekt		10	2A
	c. Kodowanie		7	2B
	d. Testowanie		10	2C
3 Budowa GUI	a. Wymagania	Paweł	10	-
	b. Projekt		13	3A
	c. Kodowanie		10	3B
	d. Testowanie		13	3C
4 Wydruki	a. Wymagania	Piotr	7	-
	b. Projekt		8	4A
	c. Kodowanie		7	4b
	d. Testowanie		7	4C
5 Wdrożenie	a. Wymagania	Paweł	2	1D;2D;3D;4D
	b. Projekt		2	5A
	c. Kodowanie		3	5B
	d. Testowanie		5	5C

Najbardziej czasochłonna ścieżka w tej sieci nazywana jest ścieżką krytyczną. Innymi słowy, ścieżka krytyczna to nieprzerwany ciąg zadań, od początku do końca sieci zadań, o najdłuższym czasie realizacji. Wszystkie zadania znajdujące się na ścieżce krytycznej są nazywane krytycznymi. Krytycznym jest więc każde zadanie, należące do tej najdłuższej ścieżki w sieci, gdyż jakiegokolwiek opóźnienie w realizacji któregośkolwiek z tych zadań wpłynie na opóźnienie całego projektu. Różnicę pomiędzy czasem wykonania tej ścieżki a czasami dla pozostałych ścieżek nazywa się rezerwą ścieżek. Z definicji, ścieżka krytyczna ma zerowy czas rezerwy. Należy zauważyć, że najkrótszy możliwy czas realizacji

projektu to czas wykonania zadań ze ścieżki krytycznej. Jak widać, może być więcej niż jedna ścieżka krytyczna. Jednocześnie, ścieżka krytyczna może zmieniać się w trakcie projektu, na przykład wydłużenie jakiegokolwiek zadania posiadającego rezerwę może wydłużyć ścieżkę tak, że stanie się ona najdłuższą a w związku z tym stanie się nową ścieżką krytyczną. Rysunek 20 przedstawia sieć PERT/CPM dla przykładowego projektu. Rysunek ten daje czytelny obraz zależności pomiędzy zadaniami. Ścieżka krytyczna złożona jest z działań związanych z interfejsem użytkownika i rozwinięciem. Określa ona minimalny czas realizacji projektu: 60 dni. Przydzielenie większej ilości zasobów, na przykład innego programisty, do tych zadań mogłoby skrócić ten czas. Z drugiej strony, czas rezerwy dla ścieżek 1, 2 i 4 wynoszą odpowiednio: 36, 31 i 33 dni.

Istnieje problem, który nie może być zobrazowany na diagramie. W istocie, cały obraz zmieniłby się, gdyby Piotr, przydzielony do obliczeń i raportów, przekroczył przewidziany na te zadania czas. Chociaż, ani **Obliczenia**, ani **Wydruki** nie są na ścieżce krytycznej, to fakt, że są one oba wykonywane przez tą samą osobę sprawia, że również stają się krytyczne. Jest to zjawisko znane jako współzależność zasobów.



Rysunek 20. Sieć PERT/CPM dla przykładowego zadania harmonogramowania

7.3 Problemy z harmonogramowaniem projektów

Jak zauważono w poprzednim punkcie, oszacowania mogą mieć znaczące rezerwy, które wyznaczają pewien margines bezpieczeństwa. Rzeczywista praktyka pokazuje jednak, że każdy margines zawsze zostanie skonsumowany, a nawet przekroczony. Istnieją co najmniej trzy przyczyny tego zjawiska: prawo Murphy'ego, prawo Parkinsona oraz tzw. syndrom studenta.

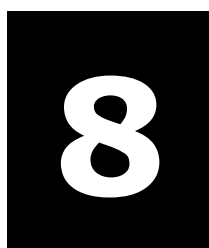
Prawo Murphy'ego. Edward A. Murphy, był inżynierem biorącym udział w eksperymentach przeprowadzanych w Jednostce Sił Powietrznych USA, mającej na celu określenie ludzkiej tolerancji na przyspieszenia. Jeden z eksperymentów wykorzystywał zestaw 16 czujników przyspieszenia przymocowywanych w różnych częściach ciała. Były 2 sposoby przymocowania takiego akcelerometru: jeden poprawny a drugi zupełnie niewłaściwy. W trakcie przeprowadzania eksperymentów, jeden z monterów metodycznie instalował wszystkie 16 urządzeń niewłaściwie. Murphy sformułował wtedy oryginalną postać swojego prawa: *"Jeśli są dwa lub więcej sposobów na zrobienie czegoś i jeden z nich prowadzi do katastrofy, to ktoś tak właśnie zrobi"* Po jakimś czasie wiele wariantów tego powiedzenia przeszło do masowego użycia, gdyż zwykle okazywały się bardzo prawdziwe. Większość z nich to wariacje na temat: *"Jeśli coś może się nie udać, to tak właśnie się stanie"*. Jedną z postaci Prawa Murphy'ego można także zastosować w kontekście tworzenia oprogramowania. Brzmiałoby ono mniej więcej tak: *„Nic nie jest realizowane zgodnie z założeniami i wszelkie oszacowania zawsze będą chybione”*.

Prawo Parkinsona. Większość twórców oprogramowania uważa że każde zadanie wykonują doskonale, ale zawsze też mogą wzbogacić je jeszcze bardziej o jakieś dodatkowe elementy, na przykład upiększenie ekranu, dodanie jakiejś ciekawej, niekoniecznie użytecznej funkcji itd. Często jednak wykorzystują oni to również jako pewien mechanizm obronny: jeśli programista zakończy pracę na długo przed upływem wyznaczonego czasu, powinien zakomunikować to przełożonemu. Istnieje jednak ryzyko, że ów przełożony, w przyszłości skróci jego kolejne oszacowania, przez co będzie wtedy pod o wiele większą presją przy

wypełnianiu kolejnych zadań i będzie się stresował, a tego nikt nie chce. Zjawisko takie znane jest powszechnie jako prawo Parkinsona i można je wyrazić za pomocą słów „*praca wypełnia cały przeznaczony na nią czas*”.

Syndrom studenta jest zjawiskiem obecnym także w dziedzinie tworzenia oprogramowania. Studenci znani są bowiem z przekładania nauki na ostatnią chwilę. Każdy student zanim weźmie się do nauki zastanawia się czemu zaczynać teraz, skoro jest jeszcze tyle czasu? Ewentualnie usprawiedliwia swoją "chęć" do nauki tym, że jest tak wiele rzeczy do zrobienia, że może należy najpierw zrobić coś innego. Wydaje się, że twórcy oprogramowania często zachowują się dokładnie tak jak studenci i odkładają swoje zadania na ostatnią chwilę, a nawet i później.

W rzeczywistości rzeczy projekty są permanentnie opóźniane. Dzieje się tak dlatego, że w większości projektów poszczególne zadania wykonywane są równolegle i na zakończenie rezultaty różnych tych zadań są integrowane, a to także może spowodować znaczne problemy.



Jakość Oprogramowania

W momencie gdy tworzymy oprogramowanie powstaje zapotrzebowanie na sprawdzenie czy i w jakim stopniu spełnia ono przyjęte założenia funkcjonalne. Do sprawdzenia kodu stosuje się różne metody inspekcji oraz testowania.

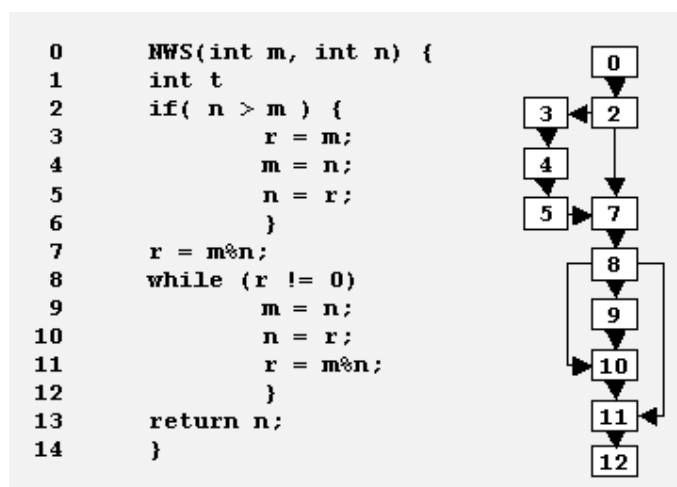
8.1 Inspekcja i walidacja

Weryfikacja potwierdza prawidłowe działanie oprogramowania i pozwala odpowiedzieć na pytanie czy oprogramowanie zostało wytworzone poprawnie. Walidacja potwierdza, że oprogramowanie działa zgodnie ze specyfikacją i tym samym z wymaganiami nałożonymi przez odbiorcę. W praktyce, takiego oszacowania można dokonać na podstawie testów. Testowanie jest procesem uruchamiania oprogramowania w kontrolowany sposób, w celu stwierdzenia czy oprogramowanie zachowuje się sposób zgodny z założeniami. Możemy mieć tu do czynienia z testami wykonawczymi, tzn. opartymi są na uruchamianiu programu. Istnieją także testy nie wykonawcze, które oparte są na przeglądach lub na analizie formalnej.

8.1.1 Testowanie strukturalne

Celem testowania strukturalnego, zwanego też metodą białej skrzynki (ang.: white box) jest weryfikacja przepływu sterowania i zwykle jest ono przeprowadzane na poziomie wnętrza danej jednostki programowej. W metodzie tej upewnia się, że wszystkie ścieżki wykonania w ramach danego modułu zostały sprawdzone co najmniej raz, wszystkie decyzje logiczne zostały sprawdzone ze wszystkimi możliwymi warunkami, pętle zostały sprawdzone przy dolnym i górnym ograniczeniu zakresu, a także że sprawdzone były wewnętrzne struktury danych. Metryką używaną w kontekście testowania strukturalnego jest złożoność cyklomatyczna. Została ona wprowadzona w latach 70-tych ubiegłego stulecia. Mierzy ona liczbę niezależnych ścieżek prowadzących poprzez dany moduł programowy. Jest ona oparta na pomysle reprezentacji przepływu sterowania w formie grafu.

Rozważmy dla przykładu poniższą funkcję w języku C, implementującą algorytm znajdowania największego wspólnego dzielnika dwóch liczb całkowitych n i m .



W grafie przepływu sterowania węzeł 2 reprezentuje decyzję w instrukcji **if**, gdzie **prawda** prowadzi do węzła 3, a **fałsz** - do węzła 7. Decyzja w pętli **while** jest reprezentowana w węźle 8, a linia pokazuje przepływ sterowania w górę z węzła 8 do 10. Gdy test w węźle 8 daje fałsz, wówczas sterowanie wychodzi z pętli do węzła 11, potem przechodzi do 12, następuje zwrócenie wyniku funkcji. Powrót z funkcji jest zamodelowany w węźle 12 reprezentującym koniec modułu. Złożoność cyklopatyczna CC jest zdefiniowana jako $(E + N + 2)$, gdzie E i N są liczbami krawędzi i węzłów grafu. Zatem dla przykładowego algorytmu złożoność ta wynosi 3.

Badania wykazują korelację pomiędzy złożonością cyklopatyczną modułu a jego częstością błędów a w związku z tym związanym z nimi ryzykiem i testowaniem. Podaje ona dokładną liczbę testów potrzebnych do sprawdzenia każdego punktu decyzyjnego w programie dla każdego wyniku testu. Nadmiernie złożony moduł będzie wymagał zbyt wielkiej liczby kroków testowych. Liczba ta może zostać zmniejszona do praktycznej wartości poprzez podział modułu na mniejsze, mniej złożone pod-moduły.

Tabela 6. Ocena programu na podstawie złożoność cyklopatyczna CC

CC	Ocena
1-10	prosty program, małe ryzyko
11-20	średnia złożoność, średnie ryzyko
21-50	duża złożoność, wysokie ryzyko
>50	nietestowalny, bardzo wysokie ryzyko

Tabela 6 prezentuje ocenę złożoności programu oraz ryzyko pojawienia się błędów w zależności od złożoności cyklometrycznej. Optymalne wartości tej metryki wynoszą poniżej 50. Należy jednak pamiętać, że są to wartości podane dla programowania strukturalnego, dlatego w przypadku programów obiektowych warto przyjąć nieco obniżony próg.

8.1.2 Testowanie jednostkowe

Jest to metoda testowania tworzonego oprogramowania poprzez wykonywanie testów weryfikujących poprawność działania pojedynczych elementów (jednostek) programu - np. metod, obiektów, procedur itp. Ten rodzaj testów koncentruje się na weryfikacji tych jednostek (stąd nazwa metody). Testerzy definiują dziedzinę wartości wejściowych dla kontrolowanych modułów i ignorują resztę systemu. Takie testy należą do typu testów strukturalnych. Testy jednostkowe mogą wykorzystywać programy testujące, koordynujące wejścia i wyjścia dla poszczególnych przypadków testowych i puste podprogramy zastępujące moduły niższego poziomu. Zaletą testów jednostkowych jest możliwość wykonywania na bieżąco w pełni zautomatyzowanych testów na modyfikowanych elementach modułu, co umożliwia często wychwycenie błędu natychmiast po jego pojawieniu się i szybką jego lokalizację zanim dojdzie do jego propagacji.

8.1.3 Testowanie behawioralne

Testy tego typu, określane również mianem testów czarnej skrzynki (ang.: black-box) polegają na sprawdzaniu możliwości funkcjonalnych (zgodność z wymaganiami) i zwykle są przeprowadzane na poziomie całego systemu. Ten typ testów jest wykorzystywany do wyszukiwania niepoprawnych lub pominiętych funkcji, błędów interfejsu, błędów w strukturach danych, niedostatków wydajnościowych oraz błędów inicjalizacji i zakończenia. Testowanie czarnej skrzynki jest również nazywane testowaniem funkcjonalnym lub behawioralnym albo testowaniem względem specyfikacji.

8.1.4 Testowanie integracyjne

Testowanie integracyjne (ang.: integration testing) służy do weryfikacji kombinacji różnych jednostek programowych. Polegają one na testowa-

niu interfejsów oraz integracji pomiędzy modułami i systemami. Testy integracyjne wykonywane są w celu wykrycia błędów w interfejsach i interakcjach pomiędzy integrowanymi modułami jak i błędów współpracy z innymi systemami taki jak: system operacyjny lub sprzęt komputerowy. Techniki projektowania przypadków testowych typu czarnej skrzynki są dominujące podczas integracji, chociaż w ograniczonym zakresie może być wykorzystane testowanie strukturalne. Testy integracyjne zapewniają systematyczne podejście do składania oprogramowania w system w sposób przyrostowy. Możemy wyróżnić dwa podejścia: zstępujące (ang.: top-down) i wstępujące (ang.: bottom-up).

W integracji zstępującej moduł główny wykorzystywany jest jako sterujący testami. Dla wszystkich modułów podrzędnych opracowywane są tak zwane zaślepki, które stopniowo zastępowane są rzeczywistymi modułami. Dodatkowe zaślepki są wymagane w rozwoju systemu w dół. Testy są prowadzone w miarę, jak każda z zaślepek jest zastępowana modułem rzeczywistym. Podejście to ma tę zaletę, że zasadniczy przepływ sterowania jest przetestowany z góry i zapewnia szybką demonstrację funkcjonalności zewnętrznej.

W integracji wstępującej, zestawiane jest zestaw modułów realizujących pewną określoną funkcję. Dla takiej konfiguracji tworzony jest moduł sterujący testami. W tym podejściu składa się te zestawy w większe całości, posuwaniu się w ten sposób w górę, aż na koniec następuje zintegrowanie z najwyższym modułem sterującym całością. Zaletą tego podejścia jest wczesne sprawdzenie modułów roboczych i brak konieczności tworzenia zaślepek lub innych symulacji danych.

8.1.5 Testowanie systemowe

Testowanie systemu to cała seria różnych testów, których podstawowym założeniem jest sprawdzenie poprawności działania całego produktu. Mimo że każdy test ma inny cel, całe przedsięwzięcie służy sprawdzeniu, czy wszystkie elementy systemu (oprogramowanie, sprzęt, interfejsy itd.) zostały prawidłowo zintegrowane i realizują przypisane im funkcje w każdych warunkach. Wyróżnia się cztery typy testowania systemu: testy przywracania, bezpieczeństwa, obciążeniowe i wydajnościowe.

8.1.6 Testowanie regresyjne

Celem przeprowadzania testów regresyjnych jest upewnienie się, że aplikacja działa po dokonaniu w niej modyfikacji, poprawieniu błędów lub po dodaniu nowej funkcjonalności. Problem polega na tym, że poprawa jednych błędów może z kolei powodować inne błędy -można zepsuć coś, co poprzednio działało. Oznacza to, że wiele elementów musi zostać przetestowanych ponownie. Jest to nazywane testowaniem regresyjnym. Ponowne wykonywanie wszystkich testów z wydania poprzedniego na wydaniu bieżącym wydaje się działaniem zasadniczo słusznym, lecz taka praktyka może okazać się kosztowna. Co więcej, nowe wersje oprogramowania mogą, poza korektami błędów, zawierać nową funkcjonalność, tak więc testy regresyjne mogłyby zużywać zasoby przeznaczone do testowania nowego kodu. A zatem dla zaoszczędzenia zasobów testy regresyjne powinny zostać zbalansowane z odpowiednimi testami jednostkowymi i mogą wykorzystywać pewne formy testowania zautomatyzowanego.

8.1.7 Testowanie *alfa* i *beta*

W sytuacji, gdy mamy do czynienia z wieloma klientami zupełnie niepraktyczne byłoby przeprowadzanie testów akceptacyjnych z każdym z nich. Większość twórców oprogramowania wykorzystuje procedurę zwaną testowaniem alfa i beta w celu wykrycia błędów swoich produktów. Testy alfa przeprowadzane są przez określonych przedstawicieli użytkowników przy udziale wykonawców oprogramowania. Testy beta są wykonywane przez użytkowników ale bez obecności wykonawców.

8.1.8 Testowanie zautomatyzowane

Testowanie zautomatyzowane to automatyczny proces testowania. Wymaga istnienia sformalizowanego procesu testowania. Warunkiem powstania takiego procesu jest opracowanie następujących elementów:

- szczegółowe przypadki testowe, zawierające oczekiwane wyniki;
- środowisko testowe, zawierające testową bazę danych, którą można przywrócić do znanego stanu w sposób pozwalający

powtarzać testy każdorazowo po dokonaniu modyfikacji aplikacji.

Rzeczywiste użycie i celowość zautomatyzowanych narzędzi testowania dotyczy automatyzacji testów regresyjnych. Oznacza to konieczność opracowania bazy powtarzalnych przypadków testowych. Pakiet testów jest uruchamiany za każdym razem, gdy zachodzi zmiana w aplikacji, celem weryfikacji, czy zmiany te nie przyniosły niezamierzonych skutków. Skrypt testu automatycznego jest programem. Z tego względu opracowywanie skryptów testów automatycznych musi być przedmiotem takich samych reguł i standardów, które są stosowane do tworzenia standardowego oprogramowania. Skrypty testowe muszą więc być pisane przez specjalistów znających istotę problemu. Nie ma możliwości generowania ich w sposób automatyczny.

8.1.9 Rezultaty testów

Rezultaty testów są równoprawną częścią dokumentacji projektu. Dają one pojęcie o znanych obszarach problemów, a także pozwalają odbiorcy zdecydować, czy produkt może zostać zaakceptowany i skierowany do finalnego wdrożenia produkcyjnego.

8.2 Zarządzanie jakością oprogramowania

Ścisła definicja pojęcia czy miary jakości oprogramowania jest trudna do przedstawienia. Wiele osób uważa, że jakość oprogramowania związana jest z brakiem błędów. Wiele podręczników inżynierii oprogramowania definiuje jakość oprogramowania jako prawidłową implementację specyfikacji. Co prawda taka definicja może być używana w procesie tworzenia oprogramowania, ale nie można jej zastosować jako oceny do porównywania produktów. Z tego powodu jakość oprogramowania często definiujemy jako dopasowanie produktu do jego celu.

Sytuację komplikuje fakt, że różne osoby mają różne cele. Zwykłego użytkownika końcowego będą bardziej dotyczyć zagadnienia związane z wygodą użycia niż zagadnienia wydajności. Integrator systemów będzie bardziej wyczulony na możliwości detekcją błędów i przywraca-

nia systemu do działania, zaś osoby zajmujące się utrzymaniem systemów będą zwracać uwagę na dokumentację techniczną. Przykłady te pokazują, że jakość oprogramowania nie jest pojęciem uniwersalnym, lecz zależnym od perspektywy oceniającego. Dodatkowo może ona również ewoluować w czasie.

Na przestrzeni ostatnich lat powstało wiele modeli nastawionych na poprawę procesu tworzenia oprogramowania. Wszystkie mają wspólny cel: przekształcić inżynierię oprogramowania ze żmudnej, nieplanowej, niekontrolowanej, intensywniej pracy we wspieraną różnymi technologiami, dyscyplinę inżynierską, skoncentrowaną na sprawdzonych praktykach technicznych i zarządczych, która za każdym razem dostarcza systemy wysokiej jakości, w zaplanowanym czasie i w ramach pierwotnie ustalonego budżetu.

8.2.1 Model CMM

Model CMM (ang.: Capability Maturity Model) powstał w instytucie SEI (ang. Software Engineering Institute) i jest oparty na założeniu, iż dojrzałość jest miarą możliwości oprogramowania oraz że dla osiągnięcia ciągłej poprawy procesów dużo lepiej jest wykonywać małe ewolucyjne kroki niż rewolucyjne rozwiązania (The Carnegie Mellon Software Engineering Institute (SEI), 2011). Model ten ukierunkowany jest na wskazywanie twórcom oprogramowania strategii poprawy procesów poprzez wstępne określenie ich bieżącej dojrzałości zanim nastąpi identyfikacja krytycznych zagadnień dotyczących poprawy jakości i usprawniania procesów. Model CMM dostarcza wytycznych do podjęcia tych kroków w formie 5 poziomów dojrzałości. Model CMM jest zastosowaniem znanej z zarządzania koncepcji TQM (ang.: Total Quality Management) i przeniesienia jej na dziedzinę inżynierii oprogramowania. Model CMM, (podobnie jak TQM) ma za cel przede wszystkim satysfakcję klienta (użytkownika). Model CMM wyróżnia pięć poziomów dojrzałości:

- Początkowy - charakteryzowany przez procesy tworzone na bieżąco. Sukces jest zależny od indywidualnego wysiłku.
- Powtarzalny - podstawowe procesy zarządzania projektami pozwalają śledzić koszt, harmonogram i funkcjonalność. Ta dyscyplina procesów umożliwia powtarzanie wcześniejszych sukcesów w podobnych projektach.

- Zdefiniowany - działania zarządcze i inżynierskie w procesie tworzenia oprogramowania są udokumentowane, zestandaryzowane i zintegrowane w ogólnofirmowy proces produkcji oprogramowania.
- Zarządzany - stosuje się szczegółowe metryki dotyczące zarówno produkcji oprogramowania, jak i jakości produktów
- Optymalizujący - stosowana ciągła poprawa procesów.

Korzyści, których można oczekiwać po implementacji zasad CMM, są znaczne i można je zaliczyć do dziedzin wzrostu produktywności i poprawy jakości. Model CMM miał znaczący wpływ na proces tworzenia oprogramowania i poprawę jakości w całym świecie. Wiele firm zaczęło używać CMM, opracowanych zostało także wiele standardów częściowo opartych na tej metodzie.

8.2.2 Standard ISO 9001

Międzynarodowa Organizacja Normalizacyjna ISO, opracowała serię norm ISO 9000. Norma ISO 9001 jest przeznaczona do zastosowania tam, gdzie konieczne jest zachowanie przez dostawcę zgodności z określonymi wymaganiami na wielu etapach, które obejmują projektowanie, rozwój produktu, produkcję, instalację oraz serwisowanie. Istnieje zbiór wskazówek ISO 9000-3, dotyczący stosowania normy ISO 9001 w rozwoju, dostarczaniu i utrzymywaniu oprogramowania. Podobnie jak metoda opisana w poprzednim punkcie, również standard ISO dotyczy ciągłej poprawy jakości i procesów. Oba standardy kładą nacisk na udokumentowane procedury i polityki, jak również na raporty i dokumentację w postaci dzienników zdarzeń oraz implementację procedur. Standard ISO oznacza konieczność zapobiegania (zapewnienia jakości) zamiast wykrywania (kontroli jakości). W praktyce uzyskanie certyfikatu ISO 9001 równoważny jest spełnieniu wymagań CMM.

8.2.3 Inicjatywa SPICE

Inicjatywa SPICE (ang.: Software Process Improvement and Capability Determination) jest międzynarodową, niezależną organizacją wspierającą rozwój międzynarodowej normy dotyczącej procesów tworzenia oprogramowania (ang.: International Standard for Software Process Assessment). Celem jej jest wspomaganie twórców oprogramowania

w osiąganiu znaczących przyrostów produktywności i jakości, przy równoczesnej pomocy odbiorcom, czyli użytkownikom oprogramowania, w uzyskiwaniu lepszego stosunku jakości do ceny i redukcji ryzyka związanego z dużymi projektami i zakupami dotyczącymi oprogramowania.

SPICE było w zamierzeniu rozwiązaniem którego celem było ulepszenie, zmiana lub optymalizacja procesów dla uzyskania większej efektywności oprogramowania i systemów informatycznych, dzięki którym osiągnąć można wzrost jakości produktów i produktywności. Obecnie SPICE dotyczy szacowania organizacji lub projektu dla określenia ryzyka pomyślnego zakończenia kontraktu lub dostarczenia usługi. Ramy dla szacowania zależą od architektury definiującej praktyki i procesów, które powinny zostać wykonane. Standard SPICE jest odpowiedzią na potrzeby użytkowników i producentów oprogramowania. SPICE dostarcza systematycznej i strukturalnej metody badania procesów wytwarzania oprogramowania oraz kryteriów ich oceny pozwalających na zebranie danych, które są: obiektywne, powtarzalne, porównywalne oraz użyteczne zarówno do doskonalenia procesu jak i do oceny jego przydatności. Standardy zaproponowane przez SPICE stały się podstawą dyrektywy ISO/IEC 12207.

Bibliografia

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison Wesley.

Booch, G., Rumbaugh, J., & Jacobson, I. (2001). *UML przewodnik użytkownika*. Wydawnictwa Naukowo Techniczne.

Cockburn, A. (2000). *Agile Software Development*. Cockburn-Highsmith.

J. Laurenz Eveleens, C. V. (2010). The Rise and Fall of the Chaos Report Figures. *IEEE Software* , 30-36.

PMI. (2008). *A Guide to the Project Management Body of Knowledge (PMBOK® Guide) - Fourth Edition*. PMI.

Project Management Institute. (2001). *A Guide to the Project Management Body of Knowledge*. PMI Press.

The Carnegie Mellon Software Engineering Institute (SEI). (2011). *Understanding and Leveraging a Supplier's CMMI A Guidebook* .

