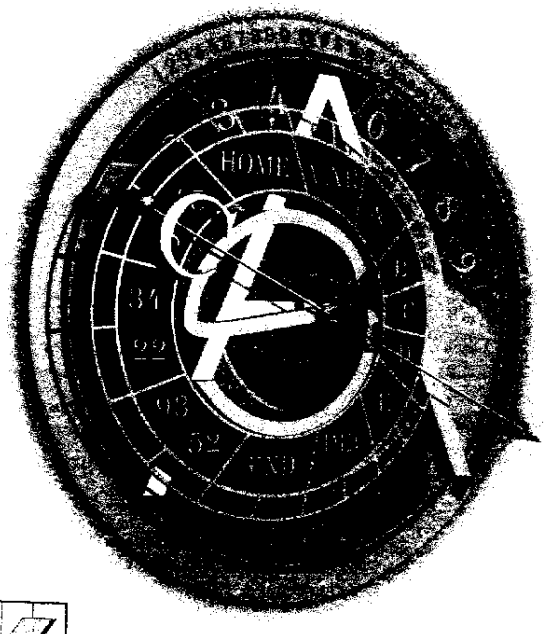
Inżynieria oprogramowania

Andrzej Jaszkiewicz



Helion

Inżynieria oprogramowania

Książka omawia podstawowe zagadnienia inżynierii oprogramowania: modele cyklu życia oprogramowania, czynności wykonywane w poszczególnych fazach przedsięwzięcia programistycznego, wykorzystanie narzędzi CASE oraz zarządzanie przedsięwzięciem programistycznym.

Przedstawione są w niej najbardziej popularne metody analizy i projektowania oprogramowania ze szczególnym uwzględnieniem metod obiektowych. Poruszane zagadnienia zilustrowane są kilkoma przykładami przewijającymi się przez całość książki.

Książka przeznaczona jest dla osób zainteresowanych profesjonalną produkcją oprogramowania. Może także służyć jako podręcznik akademicki wykorzystywany w podstawowych kursach inżynierii oprogramowania oraz projektowania systemów informatycznych.

Projekt okładki: Maciej Pasek, „ARTGRAF"
Druk: Wacław Brylewski

Copyright © HELION 1997

ISBN: 83-7197-007-2

Wydawnictwo HELION
ul. Pszczyńska 89
44-100 GLIWICE
tel./fax.: (32) 388-154,
tel: (32) 132-04-31, GSM (602) 388-154
e-mail: helion@helion.com.pl
http://www.helion.com.pl (najnowszy katalog, możliwość zamawiania książek przez Internet)

Występujące w tekście znaki są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Printed in Poland.

Spis treści

Rozdział 1.
Źródła i rola inżynierii oprogramowania

1.1. Kryzys oprogramowania

Inżynieria oprogramowania jest dziedziną informatyki, która pojawiła się w połowie lat sześćdziesiątych. Rozwój inżynierii oprogramowania poprzedziło zwrócenie uwagi na tzw. „kryzys oprogramowania". W latach pięćdziesiątych i na początku lat sześćdziesiątych tworzono wyłącznie małe programy. Wynikało to zarówno z niewielkich możliwości ówczesnych komputerów, jak i z braku zapotrzebowania na złożone oprogramowanie. Tworzono głównie oprogramowanie dla celów naukowych, wymagania były więc dość dobrze, często formalnie zdefiniowane. Co więcej, oprogramowanie było z reguły tworzone przez przyszłych użytkowników na własne potrzeby lub w ścisłej współpracy z użytkownikiem.

Sytuacja zmieniła się w połowie lat sześćdziesiątych. Rozwój sprzętu komputerowego oraz języków programowania umożliwił tworzenie znacznie bardziej złożonych systemów. Pojawili się także pierwsi programiści, tj. ludzie zawodowo zajmujący się wytwarzaniem oprogramowania. W tym okresie uświadomiono sobie również przydatność komputerów w nowych zastosowaniach, między innymi w zarządzaniu. W związku z tym podjęto szereg prób budowy złożonych systemów informatycznych, których realizacja wymagała współpracy wielu osób. Wiele z tych przedsięwzięć nie zostało nigdy ukończonych, pozostałe znacznie przekroczyły założony czas i budżet. Stało się jasne, że rozwój technik budowy oprogramowania nie nadąża za rozwojem sprzętu. To właśnie zjawisko nazwano „kryzysem oprogramowania".

„Kryzys oprogramowania" trwa zresztą do dziś. Ilustracją mogą być wyniki ankiety przeprowadzonej przez van Genuchtena (1991). W latach dziewięćdziesiątych 90% poważnych firm programistycznych w USA uważa, że często zdarzają się im opóźnienia w realizacji przedsięwzięć programistycznych. Moje obserwacje potwierdzają, że sytuacja w Polsce nie jest z pewnością pod tym względem lepsza. Oprogramowanie jest też chyba jedynym produktem technicznym, w którym błędy są powszechnie akceptowane. W 1994 roku środowisko informatyczne obiegła wiadomość o wykryciu błędu w pierwszych egzemplarzach procesora PENTIUM. Sprawa ta była przedmiotem licznych publikacji w czasopismach informatycznych oraz tematem szeregu dyskusji w internecie. Wystąpienie tego błędu, zresztą niezbyt istotnego, uznano za sytuację wyjątkową. W tym samym czasie użytkownicy programów sprzedawanych w liczbie porównywalnej do sprzedaży procesora PENTIUM i często po cenie wyższej niż ten procesor, zdają się ignorować błędy występujące w tym oprogramowaniu. Producenci programów sprzedawanych w milionach egzemplarzy nie dają zresztą praktycznie żadnej gwarancji na to, że ich produkt działa zgodnie z opisem.

Od połowy lat sześćdziesiątych do połowy lat osiemdziesiątych nie nastąpił praktycznie żaden znaczący wzrost wydajności programistów. Należy przy tym dodać, że są to dane dotyczące wyłącznie poważnych firm programistycznych, nie biorą więc one pod uwagę wpływu tzw. „rewolucji mikrokomputerowej". Trudno byłoby chyba znaleźć inną dziedzinę techniki, w której w tym okresie nie nastąpiłby wyraźny wzrost efektywności.

Jakie są przyczyny „kryzysu oprogramowania"? Liczne konferencje i dyskusje na ten temat, które odbywały się w połowie lat sześćdziesiątych, zaowocowały wskazaniem następujących jego przyczyn:

- Duża złożoność systemów informatycznych.
- Niepowtarzalność poszczególnych przedsięwzięć.
- Nieprzejrzystość procesu budowy oprogramowania, tj. fakt trudności w ocenie stopnia zaawansowania prac. Niewątpliwie najgorszym sposobem oceny postępów jest pytanie się programistów o ocenę stopnia zaawansowania. Jeśli po miesiącu uzyskany odpowiedź, że prace są zaawansowane w 90%, można się spodziewać, że przedsięwzięcie potrwa jeszcze cały rok.
- Pozorna łatwość wytwarzania i dokonywania poprawek w oprogramowaniu. Narzędzia pozwalające tworzyć nawet całkiem duże programy są stosunkowo tanie. Każdy, kto korzystając z takich narzędzi w ciągu jednego dnia napisał, uruchomił i przetestował program liczący 100 linii, może sądzić, że w ciągu 10 dni opracuje program liczący 1000 linii, w ciągu 100 dni program liczący 10.000 linii, a dziesięć takich osób w ciągu stu dni opracuje program liczący 100.000 linii. Tę przyczynę „kryzysu oprogramowania" ciekawie analizuje Baber (1989).

Rozmaite propozycje wyjścia z „kryzysu oprogramowania" zaowocowały powstaniem nowego działu informatyki — inżynierii oprogramowania.

1.2. Zakres inżynierii oprogramowania

Inżynierię oprogramowania można zdefiniować jako *wiedzę techniczną*, dotyczącą wszystkich *faz cyklu życia oprogramowania*, której celem jest uzyskanie wysokiej jakości *produktu — oprogramowania*. Definicja ta zwraca uwagę na najistotniejsze cechy tej dziedziny, które omówione są poniżej.

Inżynieria oprogramowania traktuje oprogramowanie jako produkt. Ogólne kryteria jakości oprogramowania są zbliżone do tego typu kryteriów definiowanych dla innych wyrobów. Dobre oprogramowanie powinno więc być

- zgodne z wymaganiami użytkownika,
- niezawodne,
- efektywne,
- łatwe w konserwacji,
- ergonomiczne.

Produkcja oprogramowania traktowana jest jako proces składający się z szeregu faz, począwszy od momentu, kiedy podejmowane są strategiczne decyzje o sposobie realizacji przedsięwzięcia, aż do okresu eksploatacji, w którym oprogramowanie jest wykorzystywane przez użytkowników. Implementacja, czyli kodowanie, jest tylko jednym z etapów budowy oprogramowania. Sukces całego przedsięwzięcia programistycznego zależy od jakości pracy na wszystkich etapach jego realizacji.

Inżynieria oprogramowania jest wreszcie wiedzą techniczną a nie teoretyczną nauką. Jej metody, techniki i narzędzia powstają i są rozwijane przede wszystkim w oparciu o praktyczne doświadczenie i weryfikowane są podczas ich praktycznego stosowania. Z drugiej strony, dziedzina ta wykorzystuje oczywiście dorobek wielu nauk, np. teorii programowania, sztucznej inteligencji, badań operacyjnych, psychologii i nauki o zarządzaniu.

Od początku istnienia, w ramach inżynierii oprogramowania występowały dwa nurty:

- *nurt formalny*, który postuluje jak najszersze stosowanie metod formalnych: formalnych języków specyfikacji, formalnych transformacji, formalnych dowodów poprawności,
- *nurt praktyczny*, który opiera się na stosowaniu notacji nie w pełni sformalizowanych, w dużej mierze graficznych, oraz proponuje metody, w których dużą rolę odgrywa wiedza i doświadczenie ludzkie.

O ile dorobek nurtu praktycznego jest obecnie powszechnie wykorzystywany w firmach programistycznych, o tyle nurt formalny ma raczej znaczenie teoretyczne i ciągle jest daleki od poważnych rzeczywistych zastosowań. Niniejsza książka koncentruje się prawie wyłącznie na nurcie praktycznym.

Warto też zwrócić uwagę na dwa skrajne punkty widzenia na stosunek inżynierii oprogramowania do tradycyjnych technik programistycznych.

Tradycyjny punkt widzenia uznaje pojawienie się inżynierii oprogramowania za naturalny etap rozwoju technik programowania. Jego zwolennicy uważają, że w pewnej fazie rozwoju informatyki konieczne stało się bardziej ogólne spojrzenie na tworzone systemy — spojrzenie abstrahujące od poziomu kodu w konkretnym języku programowania.

Punkt widzenia, który można nazwać *rewolucyjnym*, uważa inżynierię oprogramowania za całkowite przeciwieństwo tradycyjnych podejść do tworzenia oprogramowania. Punktem wyjścia jest konieczność zaspokojenia potrzeb użytkownika. Praca nad oprogramowaniem rozpoczyna się więc od modelowania i projektowania użytecznego systemu, na dalszym etapie rozważana jest sprawa jego implementacji. Inżynieria oprogramowania wymaga więc myślenia przede wszystkim w kategoriach zastosowania a nie w kategoriach kodu. Skrajni zwolennicy tej tezy uważają na przykład, że kształcenie informatyków powinno rozpoczynać się od nauki inżynierii oprogramowania, a dopiero później nauki konkretnych języków i środowisk programowania.

Jest oczywiste, że oba punkty widzenia są częściowo słuszne i wskazują na różne czynniki, które rzeczywiście wpływały na rozwój inżynierii oprogramowania.

Co oferuje inżynieria oprogramowania jego profesjonalnym producentom? W jaki sposób pozwala ona przynajmniej częściowo wyjść z „kryzysu oprogramowania"?

Wymienione powyżej przyczyny „kryzysu oprogramowania" wskazują także na sposoby walki z tym zjawiskiem. Należy więc:

- stosować techniki i narzędzia ułatwiające pracę nad złożonymi systemami,
- korzystać z metod wspomagających analizę nieznanych problemów oraz ułatwiających wykorzystywanie wcześniejszych doświadczeń,
- usystematyzować proces wytwarzania oprogramowania, tak aby ułatwić jego planowanie i monitorowanie,
- wytworzyć przekonanie zarówno wśród producentów, jak i nabywców oprogramowania, że budowa dużego systemu o wysokiej jakości jest zadaniem wymagającym w pełni profesjonalnego podejścia.

Inżynieria oprogramowania obejmuje więc między innymi takie zagadnienia jak:

- sposoby prowadzenia przedsięwzięć informatycznych,
- techniki planowania, szacowania kosztów, harmonogramowania i monitorowania przedsięwzięć programistycznych,
- metody analizy i projektowania systemów,
- techniki zwiększania niezawodności oprogramowania,
- sposoby testowania systemów i szacowania niezawodności,
- sposoby przygotowywania dokumentacji technicznej i użytkowej,

- procedury kontroli jakości,
- metody redukcji kosztów konserwacji (usuwania błędów, modyfikacji i rozszerzeń),
- techniki pracy zespołowej i czynniki psychologiczne wpływające na efektywność pracy.

Jak wspomniano powyżej „kryzys oprogramowania" nie został do dzisiaj w pełni zażegnany. W ostatnich latach, dzięki praktycznemu stosowaniu metod inżynierii oprogramowania, widoczne są jednak wyraźne symptomy postępu w zakresie produkcji oprogramowania. Z sukcesem realizowane są przedsięwzięcia wymagające pracy nawet ponad tysiąca osób, wiele złożonych systemów osiąga wysoki poziom niezawodności (w wypadku systemów, których nabywcy rzeczywiście takiej wysokiej niezawodności wymagają), od końca lat osiemdziesiątych następuje wyraźny wzrost wydajności pracy.

Jakie są czynniki wpływające na sukces inżynierii oprogramowania? Jej sukces w ostatnich latach wiąże się z:

- rozwojem metodyki,
- rozwojem narzędzi,
- edukacją.

Inżynieria oprogramowania pojawiła się już trzydzieści lat temu. O tego czasu nastąpił jednak znaczny jej rozwój. Wiele metod, np. obiektowych, powstało dopiero w ostatnich latach. Obecny rozwój metodyki pozwala na znaczną poprawę jakości i efektywności pracy, chociaż należy się spodziewać dalszych znaczących postępów.

Praktyczne stosowanie metod inżynierii oprogramowania bez wykorzystania specjalizowanych narzędzi, może być bardzo czasochłonne. Narzut czasu niezbędny na przykład na opracowanie szeregu diagramów i raportów w fazach analizy i projektowania, może zniweczyć potencjalne korzyści. Praktycznym rozwiązaniem tego typu problemów są narzędzia CASE — computer assisted/aided software/system engineering (komputerowo wspomagana inżynieria oprogramowania/systemów). Pojawienie się stosunkowo tanich i funkcjonalnych narzędzi CASE przypada na ostatnie lata i nadal następuje ich burzliwy rozwój.

Warunkiem sukcesu inżynierii oprogramowania jest oczywiście także znajomość metodyki i narzędzi przez osoby zaangażowane w produkcję oprogramowania. W krajach anglosaskich już od około dziesięciu lat dziedzina ta jest jednym z głównych przedmiotów w nauczaniu informatyki na poziomie uniwersyteckim. Uczelnie polskie są pod tym względem opóźnione o co najmniej kilka lat. Co więcej, na niektórych uczelniach pod hasłem inżynierii oprogramowania wykłada się na przykład teoretyczne podstawy programowania. Sytuacja zaczyna się zmieniać i obecnie kilka polskich uczelni poważnie zaangażowało się w nauczanie inżynierii oprogramowania i prowadzi praktyczne zajęcia z wykorzystaniem narzędzi CASE. Na szerokie pojawienie się wykształconej kadry inżynierów oprogramowania na polskim rynku informatycznym należy jednak poczekać jeszcze kilka lat.

1.3. Narzędzia CASE

Narzędzia CASE pełnią w pracy inżyniera oprogramowania podobną rolę jak narzędzia CAD/CAM w pracy inżynierów innych dziedzin. Tradycyjnie dzieli się je na narzędzia Upper-CASE i Lower-CASE. Programy Upper-CASE, czyli narzędzia wysokiego poziomu, koncentrują się na wstępnych etapach realizacji przedsięwzięcia, tj. określaniu wymagań, modelowaniu i projektowaniu systemu realizującego te wymagania. Narzędzia takie nie wspomagają bezpośrednio implementacji systemu. Widać więc tu wyraźny związek ze wspomnianym wcześniej rewolucyjnym spojrzeniem na źródła inżynierii oprogramowania. Programy Lower-CASE, czyli narzędzia niskiego poziomu, koncentrują się natomiast na fazie implementacji. Punktem wyjścia dla ich rozwoju była chęć ułatwienia programistom pracy nad złożonym oprogramowaniem. Opierają się one na obserwacji, że notacje graficzne są bardziej naturalnym sposobem prezentowania dużych programów niż tradycyjny zapis tekstowy. W tym wypadku wyraźnie widoczna jest analogia do tradycyjnego spojrzenia na źródła inżynierii oprogramowania.

Rozwój narzędzi CASE doprowadził do powstania narzędzi zintegrowanych I-CASE (Integrated-CASE), które łączą w sobie możliwości narzędzi Upper-CASE i Lower-Case. W efektywny sposób wspomagają one wykonywanie praktycznie wszystkich faz produkcji oprogramowania.

Rozdział 2. Modele cyklu życia oprogramowania

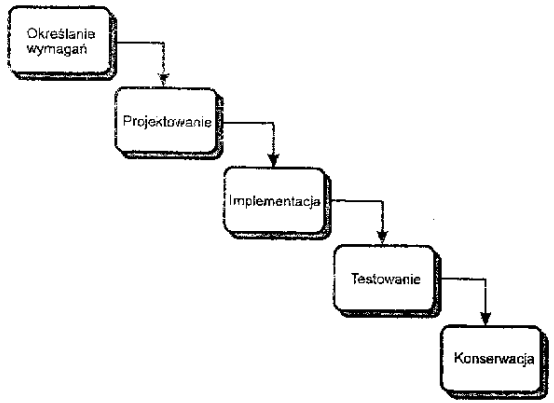
Jak wspomniano wcześniej produkcja oraz eksploatacja oprogramowania jest pewnym procesem, który powinien być realizowany w systematyczny sposób. Jak jednak dokładnie powinien wyglądać ten proces? W odpowiedzi na to pytanie proponuje się szereg tak zwanych modeli cyklu życia oprogramowania. Modele te wprowadzają pewne fazy życia oprogramowania, określają czynności wykonywane w poszczególnych fazach oraz ustalają kolejność ich realizacji. Modele cyklu życia oprogramowania pozwalają uporządkować przebieg prac, ułatwiają planowanie zadań oraz monitorowanie przebiegu ich realizacji.

2.1. Model kaskadowy

Model kaskadowy (ang. *waterfall*), zwany też modelem wodospadu lub liniowym, jest klasycznym modelem cyklu życia oprogramowania. Jest to model, który został zaproponowany poprzez analogię do sposobu prowadzenia przedsięwzięć w innych dziedzinach inżynierii, na przykład budownictwie. Budowa mostu rozpoczyna się od określenia głównych zadań jakie ma on spełniać, a następnie szczegółowego określenia wymagań, które zapewnią realizację tych zadań. Kolejnym krokiem jest wykonanie szczegółowego projektu konstrukcji. Dalej następują faza budowy oraz testowania wybudowanego mostu. Ostatnim etapem jest faza konserwacji.

Model kaskadowy cyklu życia oprogramowania wprowadza analogiczne fazy (rys. 2.1):

- *fazę określania wymagań*, w której określane są cele oraz szczegółowe wymagania wobec tworzonego systemu,



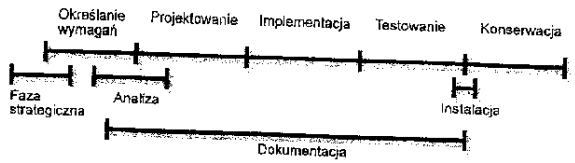
Rysunek 2.1. Kaskadowy model cyklu życia oprogramowania

- *fazę projektowania* (ang. *design*), w której powstaje szczegółowy projekt systemu spełniającego ustalone wcześniej wymagania,
- *fazę implementacji/kodowania* (ang. *implementation/coding*) oraz *testowania modułów*, w której projekt zostaje zaimplementowany w konkretnym środowisku programistycznym oraz wykonywane są testy poszczególnych modułów,
- *fazę testowania*, w której następuje integracja poszczególnych modułów połączona z testowaniem poszczególnych podsystemów oraz całego oprogramowania,
- *fazę konserwacji*, w której oprogramowanie jest wykorzystywane przez użytkownika(ów), a producent dokonuje konserwacji oprogramowania — wykonuje modyfikacje polegające na usuwaniu błędów, zmianach i rozszerzaniu funkcji systemu.

Podział całego cyklu życia na konkretne fazy proponowany przez różnych autorów może różnić się częściowo od przedstawionego powyżej. Podział ten może być mniej lub bardziej szczegółowy. Najistotniejszą cechą tego modelu jest spojrzenie na cykl życia oprogramowania jako na proces liniowy, w którym poszczególne fazy występują sekwencyjnie jedna po drugiej.

W modelu kaskadowym często wyróżnia się pewne dodatkowe fazy — zaprezentowane na rysunku 2.2 — które częściowo nakładają się na fazy wymienione powyżej:

- *Faza strategiczna* (ang. *strategy*) wykonywana przed formalnym podjęciem decyzji o realizacji przedsięwzięcia. W tej fazie podejmowane są pewne strategiczne decyzje dotyczące dalszych etapów prac. Faza ta wymaga oczywiście przynajmniej ogólnego określenia wymagań,
- *Faza analizy* (ang. *analysis*), w której budowany jest logiczny model systemu.



Rysunek 2.2. Dodatkowe fazy wyróżniane w modelu kaskadowym

- *Faza dokumentacji*, w której wytwarzana jest dokumentacja użytkownika. Opracowywanie dokumentacji przebiega równolegle z produkcją oprogramowania. Faza to może rozpocząć się już w trakcie określania wymagań. Sugeruje się nawet, że podręcznik użytkownika dla przyszłego systemu jest dobrym dokumentem opisującym wymagania. Ostatnie zmiany w dokumentacji dokonywane są w fazie instalacji.
- *Faza instalacji*, w której następuje przekazanie systemu użytkownikowi.

Poszczególne fazy tego modelu omawiane są szczegółowo w dalszej części książki.

Jakie są podstawowe zalety oraz wady modelu kaskadowego? Zalety wiążą się przede wszystkim z łatwością zarządzania przedsięwzięciem. Model ten ułatwia planowanie, harmonogramowanie oraz monitorowanie przedsięwzięcia.

Wady tego modelu to:

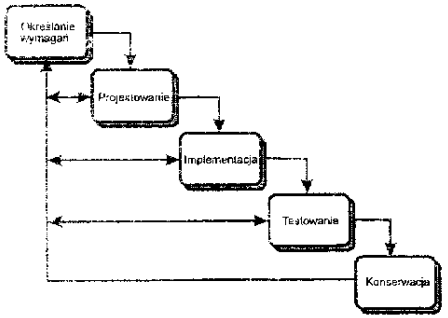
- *Narzucenie twórcom oprogramowania ścisłej kolejności wykonywania prac.* Osoby pracujące nad oprogramowaniem preferują mniej rygorystyczny styl pracy.
- *Wysoki koszt błędów popełnionych we wstępnych fazach.* Błędy popełnione w fazie określania wymagań zostaną najprawdopodobniej wykryte dopiero w fazie testowania lub konserwacji. Ich koszt może o rzędy wielkości przewyższać koszt błędów popełnionych np. w fazie implementacji.
- *Długa przerwa w kontaktach z klientem.* Faza strategiczna oraz określania wymagań i analizy realizowane są w ścisłej współpracy z klientem (w przypadku oprogramowania wykonywanego na zamówienie). Projektowanie, implementacja i w dużej mierze testowanie wykonywane są wyłącznie przez firmę programistyczną. Pojawia się więc ryzyko zmniejszenia zainteresowania klienta przedsięwzięciem w czasie gdy nie bierze on bezpośredniego udziału w jego realizacji.

Na temat praktycznej przydatności modelu kaskadowego wypowiada się skrajnie różne opinie. Z jednej strony twierdzi się, że praktycznie żadne rzeczywiste przedsięwzięcie programistyczne nie zostało zrealizowane zgodnie z tym modelem. Inni autorzy twierdzą, że zdecydowana większość rzeczywistych przedsięwzięć przebiega zgodnie z modelem kaskadowym. Ta różnica zdań wynika z faktu różnej interpretacji tego modelu.

Interpretacja ścisła traktuje poszczególne fazy jako konkretne okresy realizacji przedsięwzięcia, okresy, które nie nakładają się na siebie i są wykonywane ściśle sekwencyjnie bez żadnych iteracji.

ta interpretacja ogólna zakłada, że poszczególne fazy mają znaczenie bardziej koncepcyjne, poszczególne fragmenty systemu mogą np. znajdować się w tym samym czasie na różnych etapach realizacji. Interpretacja ta dopuszcza także iteracje — nawroty do wcześniejszych faz modelu (patrz rysunek 2.3). Oczywiście także w wypadku interpretacji ścisłej dopuszcza się nawroty w wypadku wykrycia na dalszych etapach błędów popełnionych na etapach wcześniejszych. Powroty traktowane są jednak jako sytuacje wyjątkowe, których należy w miarę możliwości unikać.

Rysunek 2.3. Iteracje w modelu kaskadowym



Wymienione powyżej wady modelu kaskadowego stały się przyczyną propozycji szeregu innych modeli cyklu życia oprogramowania. Modele te, to:

- *realizacja kierowana dokumentami* (ang. *document-driven*),
- *prototypowanie* (ang. *prototyping*),
- *programowanie odkrywcze* (ang. *exploratory programming*),
- *realizacja przyrostowa* (ang. *incremental development*),
- *model spiralny* (ang. *spiral model*),
- *formalne transformacje* (ang. *format transformations*).

Modele te są omawiane w kolejnych rozdziałach. Przy ogólnej interpretacji modelu kaskadowego można jednak większość z tych modeli uznać za pewne wersje lub rozszerzenia tego modelu. W szczególności, wymienione powyżej główne fazy przedsięwzięcia programistycznego pojawiają się w większości realistycznych modeli. W związku z tym dalsza część książki omawia poszczególne fazy modelu kaskadowego, co nie oznacza w żadnym wypadku ograniczenia się wyłącznie do tego modelu.

2.2. Realizacja kierowana dokumentami

Model ten został oryginalnie zaproponowany przez armię amerykańską. Organizacja ta tradycyjnie przeznaczała duże sumy na produkcję oprogramowania. W latach osiemdziesiątych planowano znaczne zwiększenie tych wydatków w związku z realizacją programu „gwiezdnych wojen". Szacowano, że podstawowe oprogramowanie niezbędne do realizacji tego programu będzie obejmowało kilkadziesiąt milionów instrukcji kodu źródłowego.

> Ponieważ oprogramowanie dla armii amerykańskiej musiało być tworzone w języku Ada, wśród informatyków pracujących w Stanach Zjednoczonych krążyła anegdota:
>
> Jeżeli chcesz tworzyć systemy operacyjne, to programujesz w C. Jeżeli chcesz tworzyć oprogramowanie naukowe, to programujesz w Fortranie. Jeżeli chcesz tworzyć systemy inteligentne, to programujesz w Lispie lub Prologu. Jeżeli chcesz tworzyć oprogramowanie dla biznesu, to programujesz w Cobolu. Jeżeli jednak chcesz zarabiać prawdziwe pieniądze, to programujesz w Adzie.

Armia amerykańska udowodniła kilkakrotnie, że jest w stanie udanie realizować nawet bardzo złożone przedsięwzięcia (operacja „pustynna burza", umieszczenie na orbicie wielu satelitów szpiegowskich porównywalnych z teleskopem Hubbla, wiele lat przed NASA i znacznie mniejszym kosztem). Organizacja ta zaangażowała się więc mocno także w rozwój technik inżynierii oprogramowania. Jednym z rezultatów były standardy DOD STD 2167 oraz DOD STD 2167a, opisujące wymagany sposób wykonywania oprogramowania dla armii. Model cyklu życia oprogramowania opisywany w tych dokumentach nazywa się realizacją kierowaną dokumentami (ang. *document-driven*).

Model ten jest ścisłą realizacją modelu kaskadowego, składa się więc z szeregu następujących po sobie faz. Dodatkowo zakłada się, że każda faza kończy się opracowaniem szeregu dokumentów, w pełni opisujących wyniki tej fazy. Dokumenty te powinny być wystarczającą podstawą do realizacji dalszych faz. Dokumenty te są udostępniane klientowi. Dopiero po zaakceptowaniu dokumentacji przez klienta rozpoczynana jest kolejna faza.

Model ten ma takie same zalety jak ściśle interpretowany model kaskadowy, to jest łatwe planowanie, harmonogramowanie oraz monitorowanie przedsięwzięcia. Dodatkową zaletą jest teoretyczna przynajmniej możliwość przerwania realizacji przedsięwzięcia w jednej firmie i wznowienie realizacji w innej po przekazaniu jej kompletu dokumentów. Wady tego modelu są oczywiście także takie same jak modelu kaskadowego. Dodatkowe wady to:

- Duży nakład pracy na opracowanie dokumentów zgodnych ze standardem DOD STD 2167 — ponad 50% całkowitych nakładów.
- Przerwy w realizacji przedsięwzięcia niezbędne dla weryfikacji dokumentów przez klienta.

Standard DOD STD 2167 w uproszczonej wersji bywa stosowany przez firmy realizujące oprogramowanie dla innych klientów. Obecnie inne organizacje, na przykład IEEE proponują własne standardy realizacji przedsięwzięć programistycznych, które także można uznać za przykłady realizacji kierowanej dokumentami.

2.3. Prototypowanie

Jedną z wymienionych wcześniej wad modelu kaskadowego jest duży koszt błędów popełnionych w fazie określania wymagań. W związku z tym model ten zalecany jest dla realizacji systemów, w wypadku których określenie wymagań jest stosunkowo łatwe. Typowym przykładem są tutaj systemy informatyczne w prosty sposób automatyzujące pracę pewnych organizacji, tj. automatyzujące przechowywanie i wyszukiwanie danych oraz realizację pewnych prostych operacji. Takie systemy wykonują więc praktycznie wyłącznie funkcje, które były już realizowane w danej organizacji bez wykorzystania technologii komputerowej. Systemy informatyczne pozwalają jednak także na wprowadzanie funkcji, praktycznie niewykonalnych bez stosowania komputerów, np. złożonych analiz, optymalizacji produkcji, wspomagania decyzji, czy automatycznego sterowania produkcją. Ścisłe zdefiniowanie przez klienta wymagań wobec tak nowatorskich funkcji może być bardzo trudne. Prototypowanie (ang. *prototyping*) jest modelem, którego celem jest minimalizacja ryzyka związanego z niewłaściwym określeniem wymagań.

Model ten składa się z następujących faz:

- *ogólne określenie wymagań*,
- *budowa prototypu*,
- *weryfikacja prototypu przez klienta*,
- *pełne określenie wymagań*,
- *realizacja pełnego systemu zgodnie z modelem kaskadowym*.

Głównym celem realizacji prototypu jest lepsze określenie wymagań, czyli:

- wykrycie nieporozumień pomiędzy klientem a twórcami systemu,
- wykrycie brakujących funkcji,
- wykrycie trudnych usług,
- wykrycie braków w specyfikacji wymagań.

Dodatkowe zalety budowy prototypu to:

- możliwość szybkiej demonstracji pracującej wersji systemu,
- możliwość szkolenia zanim zbudowany zostanie pełen system.

Wadą tego modelu jest oczywiście dodatkowy koszt budowy prototypu. Kolejna wada wiąże się z potencjalnym zdziwieniem klienta, który musi długo czekać i sporo płacić za końcowy system, który został „prawie całkowicie" wykonany w tak krótkim czasie.

Należy zauważyć, że prototyp nie jest częścią przyszłego pełnego systemu. Zakłada się raczej, że po dokonaniu weryfikacji przez klienta prototyp jest porzucany, a budowa pełnego systemu rozpoczyna się od nowa. W praktyce oczywiście może się okazać, że pewne fragmenty prototypu są na tyle dobre, że mogą być wykorzystane na dalszych etapach prac. Budując prototyp należy jednak przede wszystkim dążyć od jego szybkiej realizacji, z reguły kosztem jego niezawodności i efektywności.

Jakie są metody budowy prototypu? Proponowane metody, które zresztą nie wykluczają się wzajemnie, to:

- *Niepełna realizacja*. Z reguły tylko część wymagań wobec systemu jest trudna do określenia. W związku z tym proponuje się, aby w ramach prototypu ująć tylko te funkcje systemu.
- *Języki wysokiego poziomu* (Smalltalk, LISP, Prolog, języki czwartej generacji, wykonywalne języki specyfikacji formalnych).
- *Wykorzystanie gotowych komponentów*.
- *Generatory interfejsu użytkownika*. Realizacja prototypu często polega wyłącznie na realizacji interfejsu użytkownika. Przydatne są więc wszelkie narzędzia ułatwiające szybką realizację interfejsu.
- *Szybkie programowanie* (ang. *quick-and-dirty*). Prototyp może polegać na realizacji wszystkich funkcji systemu z wykorzystaniem tych samych technik, które będą stosowane przy realizacji pełnego systemu. Prace przyspiesza się jednak poprzez zaniechanie, np. procedur testowania. W związku z tym prototyp różni się od końcowego systemu przede wszystkim niezawodnością.
- *Papier*. Jest to szybka i stosunkowo lubiana przez klientów forma prototypowania interfejsu użytkownika.
- *Programowanie odkrywcze*. Model ten, omawiany w kolejnym rozdziale, jest także proponowany jako metoda budowy prototypu.

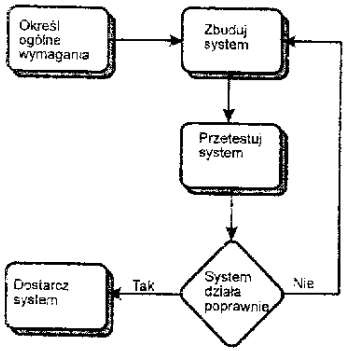
Ponieważ podstawowym zadaniem prototypu jest pomoc w lepszym określeniu wymagań, budowę i weryfikację prototypu można uznać za specyficzny sposób realizacji fazy określania wymagań tradycyjnego modelu kaskadowego. Dalszy przebieg prac jest już dokładnie taki sam jak w tym modelu.

2.4. Programowanie odkrywcze

W pewnych sytuacjach określenie wymagań klienta może być tak trudne, że nawet budowa pojedynczego prototypu nie wystarcza dla rozwiania wszelkich wątpliwości. Pro-

gramowanie odkrywcze (ang. *exploratory programming*) zalecane jest właśnie w takich wypadkach. Model ten rozpoczyna się od wstępnego, bardzo ogólnego określenia wymagań. Następnie bardzo szybko rozpoczynana jest realizacja systemu. Faza realizacji obejmuje z reguły wykonanie przynajmniej bardzo ogólnego projektu. System jest następnie weryfikowany przez klienta. Jeżeli zostanie on uznany za nieodpowiedni, budowana jest kolejna wersja systemu. Nie jest to przy tym budowa od podstaw lecz najczęściej modyfikacja poprzedniej wersji. Cykl kończy się jeżeli jedna z kolejnych wersji zadowala klienta, lub dojdzie on do wniosku, że nie jest możliwe osiągnięcie zadowalającego efektu. Schemat realizacji przedsięwzięcia zgodnie z tym modelem przedstawia rysunek 2.4.

Rysunek 2.4. Schemat programowania odkrywczego



Zaletą tego modelu jest oczywiście możliwość stosowania nawet w wypadkach dużych trudności z określeniem wymagań klienta.

Wady tego modelu to:

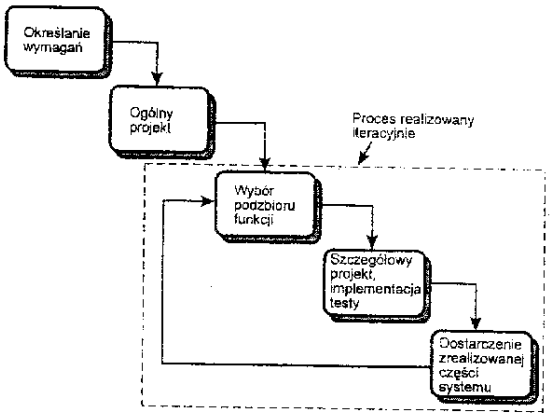
- Praktycznie niemożliwe jest zachowanie sensownej struktury systemu, nawet jeżeli na wstępie wykonano ogólny projekt. Struktura zaproponowana na wstępie na pewno zaginie w kolejnych iteracjach wprowadzania zmian do systemu. Po kilku iteracjach okazuje się, że wprowadzanie kolejnych zmian i usuwanie kolejnych błędów, powoduje dodawanie nowych błędów. W efekcie praktycznie niemożliwe jest wykonanie w ten sposób większych systemów o zadowalającej niezawodności.
- Ponieważ model ten nie obejmuje pełnego określenia wymagań, testowanie systemu może odbywać się prawie wyłącznie przy bezpośrednim udziale klienta. W wypadku innych modeli testowanie polega przede wszystkim na weryfikowaniu zgodności systemu z wcześniej sprecyzowanymi wymaganiami.

Model ten dobrze opisuje amatorski sposób tworzenia oprogramowania. Profesjonalnym producentom jest ona zalecany praktycznie wyłącznie jako sposób budowy prototypu.

2.5. Realizacja przyrostowa

Realizacja przyrostowa (ang. *incremental development*) rozpoczyna się od określenia wymagań oraz wykonania wstępnego, ogólnego projektu całości systemu. Następnie wybierany jest pewien podzbiór funkcji systemu. Dalej, zgodnie z przebiegiem modelu kaskadowego, wykonywany jest szczegółowy projekt oraz implementacja części systemu realizującej te funkcje. Po przetestowaniu zrealizowany fragment pełnego systemu może zostać dostarczony klientowi.

Rysunek 2.5. Realizacja przyrostowa



Zalety tego modelu to:

- Skrócenie przerw w kontaktach z klientem.
- Możliwość wczesnego wykorzystywania przez klienta dostarczonych fragmentów systemu.
- Możliwość elastycznego reagowania na powstałe opóźnienia. Jeżeli realizacja fragmentu systemu opóźni się, nie musi to oznaczać opóźnienia całego przedsięwzięcia. Istnieje wtedy możliwość przyspieszenia prac nad dalszymi częściami.

Wadą tego modelu jest pewien dodatkowy koszt związany niezależną realizacją fragmentów systemu. Z reguły nie jest możliwe proste wycięcie podzbioru funkcji w pełni niezależnych od pozostałych. W związku z tym może zajść konieczność implementacji tzw. szkieletów modułów, tj. modułów o interfejsie zgodnym z modułami, które znajdą się pełnym systemie lecz nie realizujących w pełni ich funkcji. Implementacja tych modułów to oczywiście pewien dodatkowy nakład pracy oraz zwiększone ryzyko nie wykrycia błędów w fazie testowania.

2.6. Montaż z gotowych elementów

Model montażu z gotowych elementów (ang. *composition of re-usable components*), zwany też programowaniem z półki (ang. *off-shell programming*), kładzie szczególny nacisk na możliwości redukcji nakładów przez maksymalne wykorzystanie podobieństw tworzonego oprogramowania do wcześniej tworzonych systemów.

Gotowe elementy mogą być wykorzystywane na różnych etapach realizacji przedsięwzięcia. Najczęściej są one wykorzystywane na etapie implementacji. Przykładem może być stosowanie:

- bibliotek,
- języków czwartej generacji, których złożone instrukcje mogą być traktowane jako odwołania do wbudowanych bibliotek,
- pełnych aplikacji, np. wykorzystanie przeglądarki plików pomocy w systemie MS Windows.

Obecnie jednak coraz częściej mówi się także o wykorzystywaniu gotowych elementów na etapie analizy i projektowania. Narzędzia CASE zdecydowanie ułatwiają wykorzystanie modeli i projektów opracowanych w poprzednich przedsięwzięciach. Pewne firmy twierdzą nawet, że są wstanie ponownie wykorzystać do 90% poprzednich wyników. Oczywiście stopień ponownego wykorzystania zależy mocno od podobieństwa poszczególnych przedsięwzięć. Pewne firmy zaczynają oferować tak zwany *design-ware*. Są to gotowe modele funkcjonowania typowego banku, firmy ubezpieczeniowej, urzędu pewnego typu. Takie modele i projekty są oczywiście rozprowadzane w formacie konkretnych narzędzi CASE. Nie zetknąłem się do tej pory z tego typu komercyjną ofertą polskich firm, spotkałem się jednak z potencjalnym przynajmniej zainteresowaniem zakupem tego typu gotowych modeli ze strony pewnych producentów oprogramowania.

Istnieją dwie metody pozyskiwania gotowych elementów:

- zakup od zewnętrznych dostawców,
- opracowanie wyników aktualnie realizowanych przedsięwzięć tak, aby mogłyby być wykorzystane w kolejnych przedsięwzięciach.

Zalety stosowania gotowych elementów to:

- *Wysoka niezawodność*. Gotowe elementy są z reguły dobrze przetestowane, ich niezawodność jest więc większa niż nowo tworzonych elementów.
- *Zmniejszenie ryzyka*.
- *Efektywne wykorzystanie specjalistów*.
- *Narzucenie standardów*. Gotowe elementy stworzone z zachowaniem pewnych standardów narzucają zachowanie tych standardów w kolejnych przedsięwzięciach.

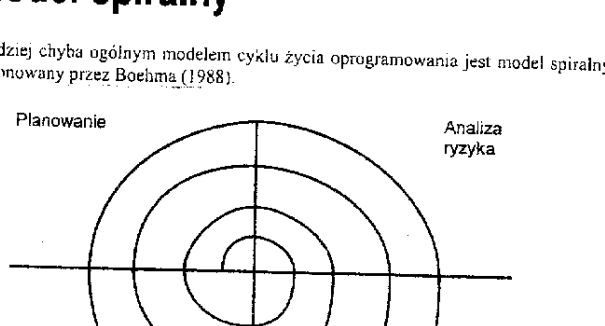
- *Potencjalna redukcja kosztów*. Koszt gotowego elementu jest z reguły znacznie mniejszy niż koszt jego pierwotnego opracowania. Pewne czynniki, które mogą jednak wpływać na wzrost kosztów wymienione są poniżej.

Wadami tego modelu są:

- *Dodatkowy koszt przygotowania elementów do ponownego wykorzystania*. Fragmenty systemu muszą być z reguły w pewien sposób przystosowane do tego celu. Jest to pewien dodatkowy nakład, który należy ponieść w trakcie realizacji przedsięwzięcia bez gwarancji, że zwróci się on w przyszłości.
- *Ryzyko uzależnienia się od dostawcy elementów*. Dostawca może zaniechać prac nad biblioteką, nie dostosować jej do nowych wersji sprzętu i oprogramowania systemowego.
- *Niedostatki narzędzi wspomagających ten rodzaj pracy*. W przypadku narzędzi CAD/CAM, które, jak wspomniano powyżej, pełnią w innych dziedzinach inżynierii rolę analogiczną do narzędzi CASE, możliwość korzystania z bibliotek gotowych elementów jest jedną z podstawowych funkcji. Narzędzia CASE do niedawna w bardzo ograniczonym zakresie wspierały ten rodzaj pracy. Najnowsze narzędzia są pod tym względem wyraźnie lepsze choć niezbędny jest dalszy rozwój tych funkcji.

2.7. Model spiralny

Najbardziej chyba ogólnym modelem cyklu życia oprogramowania jest model spiralny zaproponowany przez Boehma (1988).



Rysunek 2.6. Model spiralny

Model ten składa się z czterech głównych faz wykonywanych cyklicznie:

- *analizy ryzyka*,
- *konstrukcji*,
- *atestowania*,
- *planowania*.

W pierwszej fazie rozważane są ogólne opcje budowy nowej wersji systemu. Możliwości te są analizowane przy wzięciu pod uwagę ryzyka związanego z ich realizacją. Problematyka analizy opcji omawiana jest w kolejnym rozdziale dotyczącym fazy strategicznej realizacji przedsięwzięcia. Faza ta może obejmować także budowę prototypu.

W kolejnej fazie konstruowana jest kolejna wersja systemu w sposób zgodny z modelem kaskadowym.

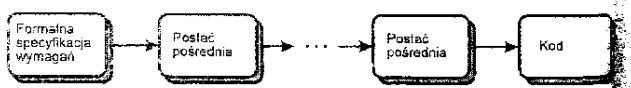
W fazie atestowania kolejna wersja systemu jest oceniana przez klienta. Jeżeli ocena nie jest w pełni pozytywna rozpoczynany jest kolejny cykl.

W fazie planowania ustalane są generalne cele produkcji kolejnej wersji systemu.

Model ten dobrze opisuje też pracę firmy wytwarzającej kolejne rynkowe wersje oprogramowania. W tym wypadku faza atestowania odpowiada eksploatacji oprogramowania przez jego użytkowników.

2.8. Formalne transformacje

Model formalnych transformacji (ang. formal transformations) jest postulowany w ramach tzw. formalnego nurtu w inżynierii oprogramowania. Zakłada on, że wymagania wobec systemu specyfikowane są w pewnym formalnym języku. Wymagania te są następnie transformowane w automatyczny sposób do pewnej postaci pośredniej bliższej kodowi w pewnym języku programowania. Postać ta podlega dalszym automatycznym transformacjom do kolejnych form coraz bliższych kodowi. Jedna z kolejnych postaci formalnych jest już na tyle bliska kodowi, że może być w automatyczny sposób przełożona na kod w konkretnym języku programowania. Wszystkie te transformacje wykonywane są bez udziału ludzi.



Rysunek 2.7. Formalne transformacje

Zaletą tego modelu jest wysoka niezawodność tak stworzonego oprogramowania. Jeżeli nie popełniono błędów na etapie specyfikowania wymagań, uzyskany w ten sposób kod wynikowy także będzie bezbłędny.

Wady tego modelu to:

- *Trudność formalnego wyspecyfikowania wymagań.* Transformacja formalnej specyfikacji wymagań do poziomu kodu przypomina w oczywisty sposób kompilację programu zapisanego w pewnym języku programowania. Język formalnej specyfikacji wymagań jest więc po prostu pewnym językiem programowania wysokiego poziomu, z reguły językiem deklaratywnym. Wyspecyfikowanie wymagań polega więc na napisaniu programu rozwiązującego pewien problem. W wypadku złożonych systemów jest to nadal bardzo trudne zadanie.
- *Mała efektywność kodu uzyskanego w ten sposób.*
- *Brak dobrze rozwiniętych, uniwersalnych języków formalnej specyfikacji wymagań.* Języki takie są z reguły opracowywane pod kątem specyficznych zastosowań. Przykładem mogą być języki formalnego modelowania danych, które mogą zostać automatycznie przetransformowane do schematu relacyjnej bazy danych. Analogiczne możliwości dają jednak znacznie częściej stosowane notacje graficzne, które dodatkowo są zdecydowanie czytelniejsze.

Model ten w chwili obecnej należy uznać za propozycję teoretyczną, która daleka jest od zastosowań w profesjonalnej produkcji oprogramowania.

Model formalnych transformacji podpowiada pewien interesujący punkt widzenia na kaskadowy model cyklu życia oprogramowania. Poszczególne fazy tego modelu można traktować jak transformacje (nieformalne) wyników poprzedniej fazy do kolejnej postaci bliższej kodowi. Cele klienta są więc transformowane do konkretnych wymagań, wymagania do modelu systemu, model do projektu implementacji, projekt do kodu. Poszczególne transformacje są jednak wykonywane przy intensywnym udziale analityków, projektantów i programistów.

2.9. Podsumowanie

Który z omówionych powyżej modeli cyklu życia oprogramowania powinien być stosowany w praktyce? Na tak postawione pytanie nie można oczywiście udzielić jednoznacznej odpowiedzi. Żaden z modeli nie może być uznany za najlepszy w każdym przypadku. Poszczególne modele różnią się pomiędzy sobą przydatnością w poszczególnych sytuacjach. Model kaskadowy jest na przykład dobrą propozycją dla firmy, która produkuje systemy o dobrze zdefiniowanych wymaganiach dla dobrze rozumianych zastosowań. Prototypowanie może okazać się lepsze w przypadku bardziej nowatorskich zastosowań.

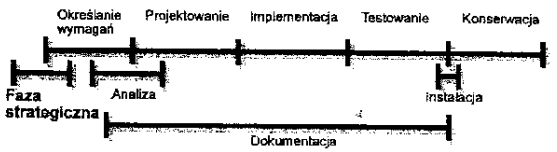
Należy też zaznaczyć, że poszczególne modele nie wykluczają się nawzajem. Budowa prototypu może być np. połączona z przyrostową realizacją pełnego systemu oraz wykorzystaniem gotowych elementów.

Wybór odpowiedniego sposobu prowadzenia konkretnego przedsięwzięcia jest jedną z najważniejszych decyzji jakie należy podjąć w fazie strategicznej omawianej w kolejnym rozdziale.

Rozdział 3.
Faza strategiczna

Faza strategiczna (ang. *strategy phase*) jest wykonywana zanim zostanie podjęta ostateczna decyzja o realizacji dalszych etapów przedsięwzięcia. W przypadku firmy produkującej oprogramowanie na zamówienie jest to więc faza, w której prowadzone są negocjacje z klientem i/lub firma bierze udział w przetargu. W przypadku firmy produkującej oprogramowanie sprzedawane rynkowo, jest to faza, w której rozważana i planowana jest produkcja nowego programu (nowej wersji programu), lecz nie została jeszcze podjęta ostateczna decyzja o rozpoczęciu prac.

Faza ta bywa także nazywana studium wykonalności (ang. *feasibility study*). Moim zdaniem nie jest to jednak odpowiednia nazwa, gdyż sugeruje ona, że jedynym celem tej fazy jest ustalenie możliwości realizacji przedsięwzięcia i podjęcie decyzji o rozpoczęciu dalszych prac. Tymczasem jest to faza, w której podejmowanych jest także szereg innych decyzji o dużym znaczeniu dla sukcesu całego przedsięwzięcia.



Rysunek 3.1. Faza strategiczna w cyklu życia oprogramowania

W fazie strategicznej wykonywane są następujące czynności:

- dokonanie serii rozmów (wywiadów) z przedstawicielami klienta,
- określenie celów przedsięwzięcia z punktu widzenia klienta,
- określenie zakresu oraz kontekstu przedsięwzięcia,
- ogólne określenie wymagań, wykonanie zgrubnej analizy i projektu systemu,
- propozycja kilku możliwych rozwiązań (sposobów realizacji systemu),

- oszacowanie kosztów oprogramowania,
- analiza rozwiązań,
- prezentacja wyników fazy strategicznej przedstawicielom klienta oraz korekcja wyników na podstawie uzyskanych uwag,
- określenie wstępnego harmonogramu przedsięwzięcia oraz struktury zespołu realizującego przedsięwzięcie,
- określenie standardów zgodnie, z którymi realizowane będzie przedsięwzięcie.

W przypadku firmy realizującej oprogramowanie na konkretne zamówienie, faza strategiczna wykonywana jest w ścisłej współpracy z klientem. Warto przy tym po stronie klienta rozróżnić *zleceniodawców* oraz *użytkowników* systemu. Ci pierwsi to osoby, których opinia będzie znacząca przy wyborze dostawcy systemu oraz przy wstępnej ocenie wyników przedsięwzięcia. Osoby te nie zawsze jednak są przyszłymi użytkownikami. W fazie strategicznej przedstawiciele producenta kontaktują się głównie ze zleceniodawcami, a więc osobami, które nie zawsze właściwie oceniają wymagania rzeczywistych użytkowników systemu.

Także firmy realizujące oprogramowanie sprzedawane rynkowo często w fazie strategicznej (a także w fazach określania wymagań i analizy) współpracują z jednym lub kilkoma potencjalnymi przyszłymi klientami. Pomocne są również badania marketingowe oraz uwagi uzyskane od użytkowników poprzednich wersji programu.

Ważnym elementem fazy strategicznej jest określenie celów przedsięwzięcia z punktu widzenia klienta. Cele te czasami mogą wydawać się oczywiste, jednak próba jasnego ich zdefiniowania często pozwala uniknąć nieporozumień pomiędzy klientem a producentem. Poza tym, nawet jeżeli cele przedsięwzięcia są jasne dla wszystkich osób zaangażowanych w fazę strategiczną, na pewno nie będą oczywiste dla osób realizujących dalsze etapy prac, w szczególności osób nie mających bezpośredniego kontaktu z klientem. Cele przedsięwzięcia powinny być ważnym punktem odniesienia w wypadku wszelkich wątpliwości pojawiających się w dalszych fazach przedsięwzięcia.

Określanie celów przedsięwzięcia zostanie zilustrowane kilkoma przykładami, które będą wykorzystywane także w dalszych częściach książki.

Program podatkowy

Firma rachunkowa zajmuje się między innymi przygotowywaniem formularzy zeznań podatkowych (PIT-ów) dotyczących podatku dochodowego dla indywidualnych podatników. Ponieważ liczba klientów korzystających z tego rodzaju usług jest duża, a w dodatku muszą oni być obsłużeni w stosunkowo krótkim czasie (w większości przypadków w marcu i kwietniu), firma widzi konieczność opracowania systemu komputerowego wspomagającego ten rodzaj działalności. Cele systemu to:

- przyspieszenie obsługi klientów — zwiększenie wydajności pracy,
- zmniejszenie ryzyka popełnienia błędów.

System informacji geograficznej — SIG

Firma programistyczna widzi możliwość sprzedaży rynkowej prostego systemu informacji geograficznej (mapy komputerowej). Miałby to być system łączący w sobie możliwość przeglądania bitowej mapy pewnego obszaru (np. mapy fizycznej, zdjęcia satelitarnego) wraz z umieszczonymi na tym tle dodatkowymi informacjami opisującymi pewne obiekty znajdujące się na prezentowanym obszarze. Cele tego systemu z punktu widzenia klienta to:

- możliwość łatwego, dialogowego projektowania mapy,
- możliwość łatwego i wygodnego przeglądania mapy.

System harmonogramowania zleceń

Przedsiębiorstwo farmaceutyczne zleciło wykonanie analizy funkcjonowania jednego z wydziałów. W ramach analizy określono między innymi funkcje (procesy) wykonywane przez wydział, wyróżniając przy tym pewne procesy krytyczne z punktu widzenia efektywności pracy wydziału. Jednym z tych procesów jest harmonogramowanie zleceń, które wydział otrzymuje z działu marketingu. Zlecenie oznacza konieczność wyprodukowania pewnej ilości konkretnego produktu, przy czym możliwe są pewne dodatkowe wymagania takie, jak obowiązek zrealizowania zlecenia przed upływem podanego terminu. W związku z tym postanowiono opracować system komputerowy wspomagający realizację tego procesu. Cele tego systemu to:

- zwiększenie wydajności pracy wydziału poprzez szybszą i efektywniejszą realizację zleceń,
- zmniejszenie opóźnień w realizowaniu zleceń,
- uwzględnienie wszelkich ograniczeń zapewniających praktyczną wykonywalność proponowanych harmonogramów,
- zapewnienie możliwości „ręcznego" modyfikowania harmonogramu,
- opracowywanie harmonogramów w formie łatwej do wykorzystania przez kadrę kierowniczą wydziału oraz automatyzacja przygotowywania zamówień dla magazynu na półprodukty.

Kolejnym ważnym zadaniem wykonywanym w fazie strategicznej jest określenie zakresu przedsięwzięcia. Z reguły opracowywany system będzie wspomagać jedynie pewien zakres działalności klienta. Właściwe określenie tego zakresu jest niezbędne dla dobrego oszacowania kosztów przedsięwzięcia. Na tym etapie klient może jednak jeszcze dokładnie nie wiedzieć jaki powinien być zakres przedsięwzięcia. Może też nie być jasne jaka część rozważanych funkcji będzie realizowana przez oprogramowanie, a jaka przez użytkowników, inne systemy lub sprzęt.

Określanie zakresu przedsięwzięcia wiąże się z opisaniem kontekstu systemu, czyli systemów zewnętrznych, z którymi współpracować będzie tworzony system. W wielu przypadkach system jest po prostu wykorzystywany przez jednego lub wielu użytkowników. Klasę użytkowników, tj. grupę osób wykorzystujących system w podobny sposób, traktuje się jako pewien system zewnętrzny. Systemem zewnętrznym może być także inny

program lub sprzęt. Czasami jako system zewnętrzny traktuje się pewne organizacje (na przykład działy firmy), które wykorzystują system. Pomija się wtedy fakt, że rzeczywiste korzystanie z systemu odbywa się za pośrednictwem pewnych osób odpowiedzialnych na przykład za wprowadzanie danych.

Poniżej opisano zakres i kontekst kilku przykładowych systemów.

Program podatkowy

Zakres przedsięwzięcia to działalność firmy rachunkowej dotycząca przygotowywania formularzy zeznań podatkowych podatku dochodowego dla indywidualnych klientów. Na tym etapie może nie być jasne, czy system powinien drukować formularz zeznania, czy tylko przygotowywać dane, które będą na formularzu umieszczane w tradycyjny sposób.

W tym wypadku użytkownik (pracownik firmy rachunkowej) jest jedynym systemem zewnętrznym, z którym system będzie współpracować.

System informacji geograficznej — SIG

Zakres przedsięwzięcia to projektowanie i przeglądanie prostej mapy komputerowej.

Systemy zewnętrzne, z którymi system będzie współpracować to:

- projektant mapy komputerowej,
- osoba przeglądająca mapę.

System harmonogramowania zleceń

Przedsięwzięcie dotyczy fragmentu funkcjonowania wydziału obejmującego przygotowywanie harmonogramu wykonywania zleceń. Niejasne jest np. czy system powinien drukować karty zadań, czy karty te mogą być nadal przygotowywane ręcznie.

Systemy zewnętrzne, z którymi system będzie współpracować to:

- system komputerowy działu marketingu,
- osoba definiująca technologiczne możliwości wydziału,
- kadra kierownicza wydziału.

Określenie celów przedsięwzięcia, jego zakresu i kontekstu to najczęściej zbyt mało, by właściwie ocenić złożoność systemu i oszacować koszt jego wykonania. Konieczne jest bardziej precyzyjne określenie wymagań, wykonanie modelu systemu oraz wstępnego projektu. Oczywiście z punktu widzenia szacowania kosztów najlepsze byłoby pełne wykonanie faz określania wymagań, analizy i projektowania. Można by wtedy bardzo dokładnie oszacować nakłady niezbędne do wykonania dalszych faz. W praktyce w fazie strategicznej czas oraz środki jakie można przeznaczyć na wykonanie tych zadań są

bardzo ograniczone. Zaleca się więc wykonanie ogólnego (zgrubnego) określania wymagań, analizy i projektu. Ważne jest przy tym, aby w ogólny sposób opisać pełen zakres systemu, tj. aby np. określić wszystkie główne wymagania oraz wykonać ogólny model całości systemu. Typowym błędem jest zbyt szczegółowe koncentrowanie się na pewnych fragmentach systemu, np. bardzo szczegółowe opisanie części wymagań z pominięciem pewnych ogólnych wymagań.

Metody określania wymagań, analizy oraz projektowania omawiane są w dalszych częściach książki.

W omawianej fazie należy podjąć szereg strategicznych decyzji dotyczących dalszego sposobu realizacji przedsięwzięcia. Decyzje te to między innymi:

- wybór modelu zgodne, z którym będzie realizowane przedsięwzięcie,
- wybór technik stosowanych w fazach analizy projektowania,
- wybór środowiska(środowisk) implementacji,
- wybór narzędzia CASE,
- określenie stopnia wykorzystania gotowych komponentów,
- podjęcie decyzji o współpracy z innymi producentami i/lub zatrudnieniu ekspertów z zewnątrz.

Bardzo często rozważa się kilka możliwych rozwiązań, tj. sposobów realizacji przedsięwzięcia. Propozycje tych rozwiązań powinny być poprzedzone określeniem ograniczeń, przy których przedsięwzięcie będzie realizowane, takich jak:

- maksymalne nakłady jakie można ponieść na realizację przedsięwzięcia,
- dostępny personel,
- dostępne narzędzia,
- ograniczenia czasowe.

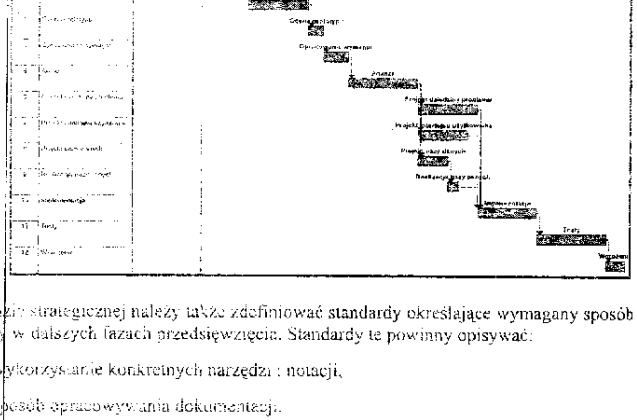
Sposoby oceny rozwiązań omawiane są w kolejnym rozdziale. Dla oceny rozwiązań niezbędne jest oczywiście oszacowanie kosztów oprogramowania dla poszczególnych propozycji. To zagadnienie jest omawiane dokładniej w sekcji 3.2.

W przypadku oprogramowania realizowanego na zamówienie, wyniki wymienionych powyżej czynności prezentowane są klientowi. Często jest to wykonywane w ramach spotkania, w którym uczestniczą przedstawiciele klienta i producenta oprogramowania. Wyniki mogą być także zaprezentowane w raporcie przekazywanym klientowi. W zdecydowanej większości przypadków klientowi proponuje się jedno rozwiązanie uznane uprzednio za najlepsze z pierwotnie rozważanych. Jeżeli po prezentacji klient przedstawi swoje uwagi, wyjaśnienia i prośby o uzupełnienie pewnych informacji, wykonywana korekcja wyników. Proces ten może być w pewnych przypadkach powtarzany kilkakrotnie, aż do uzyskania pełnej akceptacji ze strony klienta. Końcowym rezultatem jest zaakceptowanie lub odrzucenie przez klienta oferty producenta.

W przypadku firmy realizującej oprogramowanie sprzedawane rynkowo, zgromadzone wyniki są prezentowane kierownictwu wyższego szczebla, które podejmuje decyzję o rozpoczęciu lub zaniechaniu produkcji oprogramowania.

W fazie strategicznej określany jest także wstępny harmonogram przedsięwzięcia. Czynność ta polega na podziale przedsięwzięcia na pewne mniejsze zadania, określeniu terminów ich realizacji oraz zasobów niezbędnych do ich wykonania. Techniki harmonogramowania przedsięwzięć omawiane są w sekcji 13.8. Na tym etapie nie jest jeszcze możliwe wyróżnienie szczegółowych zadań. Harmonogram jest bardzo ogólny i musi być uszczegółowiany w trakcie realizacji przedsięwzięcia. Wygodnym sposobem prezentowania harmonogramów są tak zwane wykresy Gantta. Przykład takiego wykresu sporządzonego za pomocą pakietu MS Project podany jest na rysunku 3.2.

Rysunek 3.2. [illegible]



W fazie strategicznej należy także zdefiniować standardy określające wymagany sposób pracy w dalszych fazach przedsięwzięcia. Standardy te powinny opisywać:

- wykorzystanie konkretnych narzędzi i notacji,
- sposób sporządzania dokumentacji.

Dobrze zorganizowana firma powinna w dużym stopniu wykorzystywać te same standardy w różnych przedsięwzięciach. Może jednak zachodzić konieczność ich dostosowania do potrzeb konkretnego zadania. Zmiana standardów jest oczywiście konieczna, jeżeli nastąpi istotna zmiana sposobu pracy, np. stosowane są nowe techniki i/lub narzędzia. Dalsze zagadnienia dotyczące standardów omawiane są w rozdziale 13.3 omawiającym zapewnianie jakości.

3.1. Ocena rozwiązań

Jak wspomniano powyżej, wybór najlepszego sposobu realizacji przedsięwzięcia (rozwiązania) jest podstawowym warunkiem końcowego sukcesu. W fazie strategicznej rozważanych jest często kilka możliwych rozwiązań, z których następnie wybiera się jedno najlepsze. Istnieją dwa podstawowe źródła trudności w porównywaniu rozwiązań:

- wielość celów przedsięwzięcia (z punktu widzenia producenta), czyli wielość kryteriów oceny proponowanych rozwiązań,
- niepewność, tj. fakt niemożliwości precyzyjnej oceny spodziewanych rezultatów wyboru danego rozwiązania.

Kryteria stosowane do oceny rozwiązań mogą być różne w zależności od kontekstu danego przedsięwzięcia. Firma może kierować się np. takimi kryteriami jak:

- koszt,
- czas realizacji,
- niezawodność,
- stopień możliwości ponownego wykorzystania fragmentów systemu,
- przenośność na inne platformy,
- wydajność.

Wygodnym sposobem prezentowania poszczególnych rozwiązań jest zapis tabelaryczny, w którym wszystkie rozwiązania opisane są wartościami poszczególnych kryteriów. Przykładem może być tabela 3.1 zawierająca opis trzech rozwiązań.

Tabela 3.1.
Tabelaryczny zapis rozważanych rozwiązań

Rozwiązanie	*A*	*B*	*C*
Koszt [tys. zł]	120	80	175
Czas [miesiące]	33	30	36
Niezawodność [liczba błędnych wykonań/tydzień]	5	9	13
Ponowne wykorzystanie [%]	40	40	30
Przenośność [%]	90	75	30
Wydajność [transakcji/s]	0,35	0,75	1

Pierwszym krokiem powinno być usunięcie rozwiązań zdominowanych. Rozwiązanie jest zdominowane jeżeli istnieje inne rozwiązanie nie gorsze z punktu widzenia żadnego kryterium i lepsze na co najmniej jednym kryterium. Gdyby przewidywana wydajność dla rozwiązania C wynosiła 0,75 [transakcji/s] byłoby to rozwiązanie zdominowane, gdyż rozwiązanie B daje taką samą wydajność i jest lepsze pod względem innych kryteriów. Jeżeli różnica pomiędzy wydajnością 0,75 [transakcji/s] a 1 [transakcji/s] jest zdaniem kierownictwa firmy nieistotna, można rozwiązanie C uznać za „praktycznie zdominowane" i wyeliminować z dalszych rozważań. W praktyce najczęściej poszczególne rozwiązania są szczególnie dobre z punktu widzenia pewnych kryteriów, rzadko więc pojawiają się rozwiązania zdominowane.

Poszczególne kryteria oceny różnią się z reguły swoim znaczeniem. Można im więc nadać pewne wagi, które oczywiście mogą być różne w różnych przedsięwzięciach. Wagi pozwalają obliczyć łączną ocenę poszczególnych rozwiązań, będącą sumą ważoną wszystkich kryteriów. Ponieważ jednak poszczególne kryteria wyrażane są w różnych jednostkach i przyjmują wartości z zupełnie różnych zakresów nie jest sensowne dodawanie do siebie wartości tych kryteriów nawet przy uwzględnieniu wag. Konieczne jest więc znormalizowanie wartości kryteriów, tak aby wszystkie przyjmowały wartości z podobnego zakresu. Wszystkie kryteria można np. znormalizować do zakresu [0,1] gdzie 0 oznacza najgorszą wartość a 1 wartość najlepszą. Wartość 0 można przydzielić najgorszej rzeczywistej wartości danego kryterium, wartość 1 najlepszej rzeczywistej wartości, a pozostałym wartościom kryteriów proporcjonalnie obliczone wartości pośrednie. Wartości znormalizowane, po przemnożeniu przez wagi kryteriów, mogą być sumowane w celu otrzymania łącznej oceny rozwiązań. Tabela 3.2 zawiera znormalizowane wartości kryteriów dla sytuacji opisanej w tabeli 3.1, wagi poszczególnych kryteriów i łączne oceny rozwiązań. Jak widać, w tym wypadku rozwiązanie B może być uznane za najlepsze. Arkusze kalkulacyjne są wygodnymi narzędziami do wykonania tego typu obliczeń. Pozwalają one na szybką analizę wpływu zmian wartości wag lub zmian wartości pewnych kryteriów na łączną ocenę.

Suma ważona jest tylko jednym z wielu możliwych sposobów oceny opcji w obecności wielu kryteriów. Obszerna dziedzina zajmująca się tymi zagadnieniami nosi nazwę wielokryterialnego wspomagania decyzji (Roy, 1985).

Tabela 3.2.
Łączna ocena rozwiązań za pomocą sumy ważonej

Rozwiązanie	*A*	*B*	*C*	*Waga*
Koszt	0,58	1	0	3
Czas	0,5	1	0	2
Niezawodność	1	0,5	0	3
Ponowne wykorzystanie	1	1	0	1
Przenośność	1	0,75	0	1
Wydajność	0	0,62	1	1,5
Łączna ocena	**7,74**	**9,17**	**1,5**	

Niepewność jest kolejnym czynnikiem utrudniającym wybór najlepszego rozwiązania. Przykładem może być sytuacja firmy, która rozważa wykorzystanie nowego środowiska programistycznego (rozwiązanie A) lub zastosowanie dotychczasowego (rozwiązanie B). Dla każdego z tych rozwiązań oszacowano koszt optymistyczny i pesymistyczny (tabela 3.3). Jak widać rozrzut tych kosztów dla opcji B jest znacznie mniejszy.

Tabela 3.3.
Przykład niepewności w ocenie kosztu

Rozwiązanie	*A*	*B*
Pesymistyczny koszt [tys zł]	100	80
Optymistyczny koszt [tys zł]	40	65

Jeżeli nie są dostępne żadne dodatkowe informacje, osoby oceniające te rozwiązania mogą wybrać jedną z dwóch strategii:

- strategię konserwatywną (pesymistyczną), która bierze pod uwagę szacunki pesymistyczne,
- strategię optymistyczną, która bierze pod uwagę szacunki optymistyczne.

W pierwszym wypadku zostałoby wybrane rozwiązanie B, w drugim rozwiązanie A. Nie można w obiektywny sposób stwierdzić, która z tych strategii jest lepsza.

Bardziej racjonalna ocena jest możliwa jeżeli dodatkowo zostaną oszacowane tak zwane prawdopodobieństwa subiektywne zajścia poszczególnych zdarzeń pod warunkiem wybrania danego rozwiązania. W tym celu należy oszacować prawdopodobieństwa sukcesu i porażki (osiągnięcia optymistycznego i pesymistycznego kosztu) przy wyborze rozwiązań A i B. Przykładowe prawdopodobieństwa podane są w tabeli 3.4.

Tabela 3.4.
Prawdopodobieństwa subiektywne

Rozwiązanie	*A*	*B*
Prawdopodobieństwo pesymistycznego rozwiązania	0,5	0,2
Prawdopodobieństwo optymistycznego rozwiązania	0,5	0,8

Powyższe informacje pozwalają obliczyć tak zwany spodziewany koszt dla poszczególnych opcji:

Spodziewany koszt rozwiązania A = koszt pesymistyczny * prawdopodobieństwo zajścia tego zdarzenia + koszt optymistyczny * prawdopodobieństwo zajścia tego zdarzenia = = 0,5 * 100 + 0,5 * 40 = 70 [tys. zł].

Spodziewany koszt rozwiązania B = 0,2 * 80 + 0,8 * 65 = 68 [tys. zł]

Spodziewany koszt jest więc niższy dla rozwiązania B. Jeżeli jednak prawdopodobieństwo sukcesu w przypadku rozwiązania A oceniono by na 0,4 uzyskano by następujący wynik:

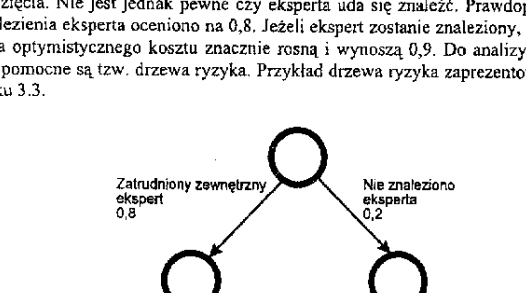
Spodziewany koszt rozwiązania A = 0,4 * 100 + 0,6 * 40 = 64 [tys. zł]

Spodziewany koszt rozwiązania B = 0,2 * 80 + 0,8 * 65 = 68 [tys. zł]

Racjonalny byłby wówczas wybór rozwiązania A.

W wielu sytuacjach należy brać pod uwagę wiele zdarzeń, które mogą zajść w wypadku wyboru pewnego rozwiązania. Firma pragnąca zastosować nowe środowisko programistyczne może na przykład rozważać zatrudnienie zewnętrznego eksperta na czas realizacji przedsięwzięcia. Nie jest jednak pewne czy eksperta uda się znaleźć. Prawdopodobieństwo znalezienia eksperta oceniono na 0,8. Jeżeli ekspert zostanie znaleziony, szanse osiągnięcia optymistycznego kosztu znacznie rosną i wynoszą 0,9. Do analizy tego typu sytuacji pomocne są tzw. drzewa ryzyka. Przykład drzewa ryzyka zaprezentowany jest na rysunku 3.3.

Rysunek 3.3. Przykład drzewa ryzyka



Wierzchołki drzewa odpowiadają sytuacjom, w których mogą zajść różne zdarzenia (można rozważać dwa lub więcej zdarzeń). Krawędzie odpowiadają zdarzeniom. Dla każdego zdarzenia podano jego prawdopodobieństwo pod warunkiem znalezienia się w sytuacji odpowiadającej wierzchołkowi, z którego wychodzi dana krawędź. Dolne wierzchołki (liście) odpowiadają możliwym scenariuszom zajścia zdarzeń. Prawdopodobieństwo zajścia danego scenariusza jest iloczynem prawdopodobieństw wszystkich krawędzi jakie do niego prowadzą. Informacje zaprezentowane na rysunku 3.3 pozwalają obliczyć spodziewany koszt rozwiązania polegającego na zastosowaniu nowego środowiska programistycznego i próbie zatrudnienia zewnętrznego eksperta:

Spodziewany koszt = 45 * 0,72 + 105 * 0,08 + 40 * 0,5 + 100 * 0,5 = 54,8 [tys. zł]

3.2. Szacowanie kosztów oprogramowania

Wstępne oszacowanie kosztów oprogramowania musi zostać wykonane dla każdego z rozważanych rozwiązań. Dla ostatecznie wybranego rozwiązania niezbędne jest wykonanie dokładniejszego oszacowania.

Na koszt oprogramowania składają się następujące główne czynniki:

- koszt sprzętu będącego częścią tworzonego systemu,
- koszt wyjazdów i szkoleń,
- koszt zakupu narzędzi,
- nakład pracy.

Trzy pierwsze czynniki są stosunkowo łatwe do oszacowania, natomiast ocena nakładów pracy niezbędnych dla zrealizowania systemu jest bardzo trudna. Dlatego też szacowanie kosztów oprogramowania jest praktycznie utożsamiane z szacowaniem nakładów pracy.

Dla oszacowania kosztów oprogramowania (nakładów pracy) stosuje się następujące techniki:

- *Modele algorytmiczne.* Techniki te wymagają opisu danego przedsięwzięcia za pomocą szeregu atrybutów liczbowych i/lub opisowych. Odpowiedni algorytm (często zwykła formuła matematyczna) daje następnie w wyniku spodziewany nakład pracy.
- *Ocena przez eksperta(ów).* Doświadczone osoby często z dużą precyzją potrafią oszacować koszt realizacji nowego systemu.
- *Ocena przez analogię.* Metoda ta wymaga dostępu do informacji o poprzednio realizowanych przedsięwzięciach. Mogą to być przedsięwzięcia wykonywane w firmie, która wykonuje szacowanie kosztów lub w innych firmach o podobnym profilu. Komercyjnie sprzedawane są np. bazy danych zawierających informacje o różnorodnych przedsięwzięciach realizowanych przez rozmaite firmy. Ogólnie rzecz biorąc metoda ta polega na wyszukaniu przedsięwzięcia (przedsięwzięć) podobnych do aktualnie rozważanego i wykorzystaniu informacji o nakładach poniesionych na ich realizację. Do tej grupy technik należy też zaliczyć stosowanie systemów automatycznego uczenia się (sieci neuronowych, systemów regułowych). System taki uczy się na podstawie pewnego zbioru przedsięwzięć, a następnie jest wykorzystywany do oceny nowych przedsięwzięć.
- *Prawo Parkinsona.* Jest to prawo, które stwierdza, że przedsięwzięcia, w tym także przedsięwzięcia programistyczne, praktycznie zawsze są wykonywane przy założonych nakładach.

- *Wycena dla wygranej* (ang. *pricing to win*). Koszt oprogramowania szacowany jest na podstawie oceny możliwości klienta(ów) oraz przewidywanych działań konkurentów. Prawo Parkinsona pozwala wierzyć, że przedsięwzięcie i tak zmieści się w założonych nakładach. Moje doświadczenie wskazuje, że jest to technika często stosowana przez firmy biorące udział w przetargach na opracowanie oprogramowania.

- *Szacowanie wstępujące*. Realizację przedsięwzięcia dzieli się mniejsze zadania, których koszt jest łatwiejszy do oszacowania. Następnie oblicza się koszt całego przedsięwzięcia. Przykładem szacowania wstępującego są techniki harmonogramowania omawiane w sekcji 13.8.

Pewnego komentarza wymaga prawo Parkinsona. Jaki jest mechanizm powodujący, że przedsięwzięcia programistyczne mają tendencję do dokładnego mieszczenia się w założonych nakładach? Moje własne doświadczenie oraz konsultacje z kierownikami firm programistycznych wskazują na następujący mechanizm. Jeżeli nakład pracy zostanie przeszacowany, dodatkowy czas zostanie bez trudu zagospodarowany przez twórców oprogramowania na czynności w niewielkim stopniu wpływające na rzeczywistą jakość końcowego produktu. Kierownik pewnego przedsięwzięcia stwierdził, że „jeżeli programista wykona w ciągu jednego dnia zadanie, na które dostał tydzień, to potrafi poświęcić pozostałe cztery dni na to, by na karcie graficznej z ośmioma kolorami uzyskać dziewiąty kolor". Jeżeli natomiast nakład pracy zostanie niedoszacowany, kierownictwo przedsięwzięcia może dążyć do zmieszczenia się planach poprzez rezygnację z części funkcji systemu i/lub przez obniżenie niezawodności oprogramowania (na przykład redukując czas trwania testów). Obniża to jednak w oczywisty sposób jakość systemu.

3.3. Algorytmiczne modele szacowania kosztów oprogramowania — model COCOMO

Stosowanie modeli algorytmicznych rozpoczyna się od opisania danego przedsięwzięcia za pomocą szeregu atrybutów. W zdecydowanej większości przypadków jednym z tych atrybutów jest rozmiar systemu, mierzony np. liczbą instrukcji kodu źródłowego. Modele takie opierają się więc na założeniu, że oszacowanie rozmiaru systemu jest znacznie łatwiejsze niż oszacowania nakładu pracy.

Proponuje się oczywiście także modele algorytmiczne, które nie opierają się na oszacowaniu liczby instrukcji kodu. Przykładem może być metoda punktów funkcyjnych (ang. *function points*). Metoda ta opiera się na oszacowaniu następujących czynników wpływających na złożoność systemu:

- dane odczytywane przez system i pobierane z systemu,

- interakcja z użytkownikiem,

- zewnętrzne interfejsy,

- pliki wykorzystywane przez system.

Jednym z częściej wykorzystywanych modeli algorytmicznych jest model COCOMO (COst COnstruction MOdel). Model ten powstał w oparciu o informacje dotyczące wielu rzeczywistych przedsięwzięć. Ponieważ jednak przedsięwzięcia te były realizowane w sposób tradycyjny, bez wykorzystania narzędzi CASE, można się spodziewać, że model ten nie jest w pełni adekwatny w przypadku współczesnych przedsięwzięć. Dlatego też nie są omawiane wszystkie szczegóły tego modelu. Z drugiej strony, znajomość czynników branych pod uwagę w tym modelu jest niewątpliwie przydatna, nawet jeżeli konkretne parametry liczbowe tego modelu nie w pełni odpowiadają przedsięwzięciom wykorzystującym narzędzia CASE. Interesująca jest także ogólna postać zależności pojawiających się w tym modelu.

Stosowanie modelu COCOMO wymaga oszacowania liczby instrukcji, z których będzie się składał system. Rozważane przedsięwzięcie jest następnie zaliczane do jednej z klas:

- *Przedsięwzięć organicznych* (ang. *organic projects*). Klasa ta obejmuje przedsięwzięcia wykonywane przez stosunkowo małe zespoły, złożone z osób o podobnym, wysokim poziomie umiejętności technicznych. Dziedzina problemu jest dość dobrze znana. Przedsięwzięcie jest wykonywane za pomocą dobrze znanych metod i narzędzi.

- *Przedsięwzięć półoderwanych* (ang. *semi-detached projects*). Członkowie zespołu różnią się stopniem zaawansowania. Pewne aspekty dziedziny problemu, część metod i narzędzi nie jest dobrze znana.

- *Przedsięwzięć osadzonych* (ang. *embeded projects*). Klasa ta obejmuje przedsięwzięcia polegające na realizacji systemów o bardzo złożonych wymaganiach (np. wymagających ścisłej współpracy ze sprzętem). Dziedzina problemu, stosowane metody i narzędzia są w dużej mierze nieznane. Większość członków zespołu nie ma doświadczenia w realizacji podobnych zadań.

Podstawowe wyrażenie stosowane w tym modelu pozwalające wyznaczyć nakład pracy to:

$$\text{Nakład [osobomiesiące]} = A\,(\text{KDSI})^b,$$

gdzie KDSI oznacza tysiąc instrukcji kodu źródłowego (ang. thousand of delivered source code instructions). KDSI nie obejmuje więc fragmentów kodu, które nie zostały ostatecznie wykorzystane w końcowym systemie, mimo że ich opracowanie mogło wymagać sporych nakładów.

Wartości stałych A i b zależą od klasy, do której zaliczono przedsięwzięcie:

- przedsięwzięcia organiczne:

$$\text{Nakład} = 2{,}4\,(\text{KDSI})^{1{,}05}$$

- przedsięwzięcia półoderwane:

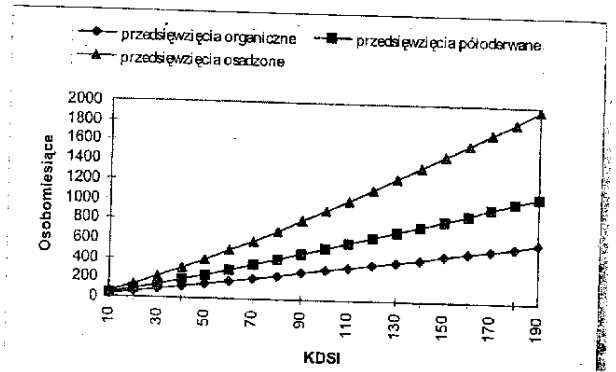
  Nakład = 3 (KDSI)^{1,12}

- przedsięwzięcia osadzone:

  Nakład = 3,6 (KDSI)^{1,20}

Rysunek 3.4. w sposób graficzny obrazuje zależność nakładów od rozmiaru systemu dl poszczególnych klas przedsięwzięć. Zależność ta jest nieliniowa, nakłady rosną wię szybciej niż proporcjonalnie do liczby instrukcji kodu źródłowego. Wzrost ten jes szczególnie szybki dla trudnych przedsięwzięć. Dla niewielkich, stosunkowo prostyc przedsięwzięć zależność ta jest bliska liniowej.

Rysunek 3.4. Oszacowania nakładów w modelu COCOMO



Model COCOMO zakłada także, że znając nakład pracy można oszacować czas rea zacji przedsięwzięcia, z czego wynika oczywiście przybliżona wielkość zespołu, któ powinien realizować przedsięwzięcie. Wynika to z obserwacji, że dla każdego prze sięwzięcia istnieje optymalna wielkość zespołu wykonawców. Zwiększenie liczby osó może nawet spowodować wydłużenie czasu realizacji. Proponowane wyrażenia, pozwa lające oszacować czas trwania przedsięwzięć, są następujące:

- przedsięwzięcia organiczne:

  Czas [miesiące] = 2,5 (Nakład)^{0,32},

- przedsięwzięcia półoderwane:

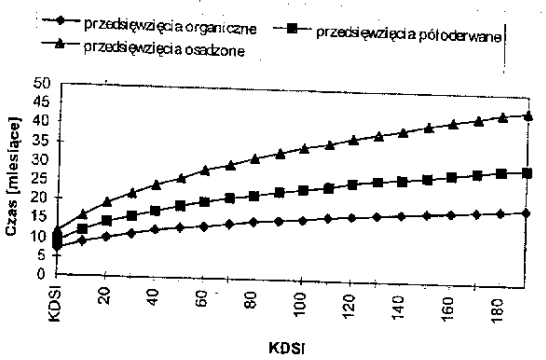
  Czas [miesiące] = 2,5 (Nakład)^{0,35},

- przedsięwzięcia osadzone:

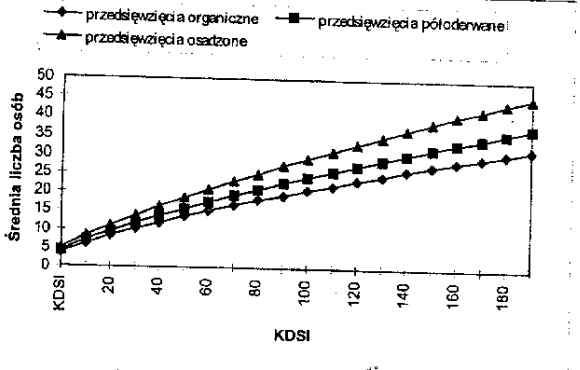
  Czas [miesiące] = 2,5 (Nakład)^{0,38}.

Zależności te są w sposób graficzny zobrazowane na rysunkach 3.5 i 3.6.

Rysunek 3.5. Oszacowania czasu trwania przedsięwzięć w modelu COCOMO



Rysunek 3.6. Oszacowania średniej liczby osób niezbędnych do realizacji przedsięwzięcia w modelu COCOMO

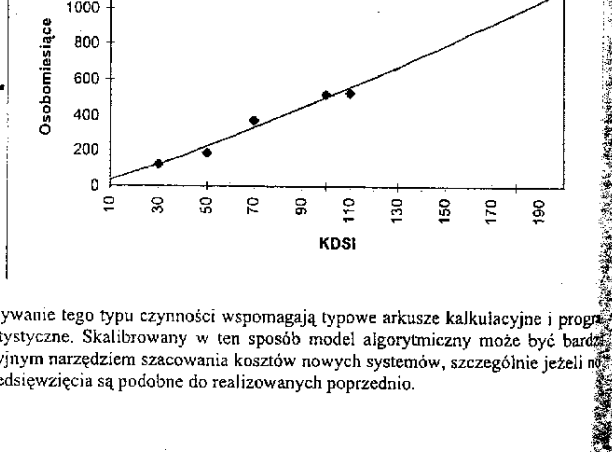


Przedstawione powyżej zależności to podstawowy model COCOMO. Otrzymane w ten sposób oszacowania powinny zostać skorygowane za pomocą tzw. czynników modyfikujących. Opierają się one na oszacowaniu pewnych dodatkowych atrybutów opisujących przedsięwzięcie. Z każdym atrybutem wiąże się pewien czynnik modyfikujący nieznacznie mniejszy lub większy od jedności. Oszacowane za pomocą modelu podstawowego nakłady mnoży się następnie przez czynniki modyfikujące. Wartość czynnika większa od jedności wpływa więc na wzrost nakładów, a wartość mniejsza od jedności na ich zmniejszenie. Czynniki modyfikujące tworzy się biorąc pod uwagę następujące atrybuty przedsięwzięcia:

- wymagania wobec niezawodności systemu,

- rozmiar bazy danych w stosunku do rozmiaru kodu,
- złożoność systemu, tj. złożoność struktur danych, złożoność algorytmów, komunikacja z innymi systemami, stosowanie obliczeń równoległych,
- wymagania co do wydajności systemu,
- ograniczenia pamięci,
- zmienność maszyny wirtualnej, tj. zmienność sprzętu i oprogramowania systemowego tworzących środowisko pracy systemu.

Liczne obserwacje wskazują na bardzo znaczące różnice w wydajności poszczególnych firm programistycznych. Podobne przedsięwzięcia mogą być realizowane przez pewne firmy przy dziesięciokrotnie niższych nakładach. Powstaje więc pytanie, czy przy tak znacznym rozrzucie wydajności możliwe jest sensowne stosowanie modelu COCOMO i innych modeli algorytmicznych? Czy nie są to modele opisujące wyłącznie jakieś wyidealizowane średnie przedsiębiorstwo? Praktyczne wykorzystanie modeli algorytmicznych wymaga ich dostrojenia (kalibracji) do sytuacji danej firmy. Do tego celu niezbędne są dane liczbowe dotyczące poprzednich przedsięwzięć. Na ich podstawie można — stosując na przykład metodę najmniejszych kwadratów — znaleźć stałe A i b, dające przewidywania maksymalnie zgodne z obserwacjami (patrz rysunek 3.7).



Rysunek 3.7. Dostrajanie modelu algorytmicznego na podstawie danych o poprzednich przedsięwzięciach

Wykonywanie tego typu czynności wspomagają typowe arkusze kalkulacyjne i programy statystyczne. Skalibrowany w ten sposób model algorytmiczny może być bardzo precyzyjnym narzędziem szacowania kosztów nowych systemów, szczególnie jeżeli nowe przedsięwzięcia są podobne do realizowanych poprzednio.

3.4. Podsumowanie

3.4.1. Kluczowe czynniki sukcesu

Zakończona sukcesem realizacja fazy strategicznej zależy oczywiście od wielu czynników. Poniżej wymieniono najistotniejsze z nich, czyli tzw. kluczowe czynniki sukcesu:

- *Szybkość pracy.* Szczególnie w przypadku firm realizujących oprogramowanie na zamówienie, startujących w przetargach, opóźnienia w realizacji tej fazy mogą zaprzepaścić szansę na uzyskanie zamówienia. Niezależnie od rodzaju tworzonego systemu, firma programistyczna z reguły nie może sobie pozwolić na przeznaczenie zbyt dużych nakładów na tę fazę. Wymaga to więc wykonania tej fazy przez stosunkowo niewielką liczbę osób w krótkim czasie.
- *Zaangażowanie kluczowych osób ze strony klienta.* Brak akceptacji dla sposobu realizacji przedsięwzięcia ze strony pewnych kluczowych osób może praktycznie uniemożliwić jego przyszły sukces.
- *Uchwycenie (ogólne) całości systemu.* Podstawowym błędem popełnianym w tej fazie jest zbyt szybka koncentracja na pewnych fragmentach systemu. Niemożliwe jest w takiej sytuacji właściwe oszacowanie kosztów wykonania systemu. Łatwo jest także przeoczyć szczególnie trudne fragmenty systemu.

3.4.2. Podstawowe rezultaty fazy strategicznej

Podstawowe rezultaty fazy strategicznej to:

- udostępniany klientowi raport obejmujący:
  - definicję celów przedsięwzięcia,
  - opis zakresu przedsięwzięcia,
  - opis systemów zewnętrznych, z którymi system będzie współpracować,
  - ogólny opis wymagań,
  - ogólny model systemu,
  - opis proponowanego rozwiązania,
  - oszacowanie kosztów,
  - wstępny harmonogram prac,
- raport oceny rozwiązań, zawierający informacje o rozważanych rozwiązaniach oraz przyczynach wyboru jednego z nich,
- opis wymaganych zasobów — pracownicy, oprogramowanie, sprzęt,
- definicje standardów,
- harmonogram fazy analizy.

3.4.3. Narzędzia CASE w fazie strategicznej

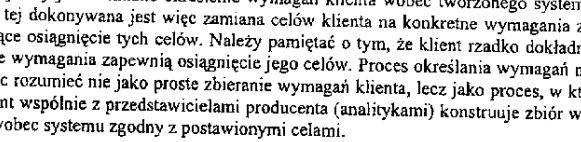
Ponieważ faza strategiczna obejmuje ogólne określenie wymagań, wstępną analizę oraz projekt, wykorzystywane są w tej fazie te same narzędzia, które będą wykorzystywane w kolejnych fazach. Narzędzia te zostaną omówione dokładniej przy szczegółowym omawianiu tych faz.

Na rynku dostępne są programy wspomagające stosowanie rozmaitych metod szacowania kosztów oprogramowania. Mogą one być częściami większych systemów CASE lub niezależnymi programami.

W fazie strategicznej wykorzystuje się także narzędzia harmonogramowania. Pozwalają one np. znaleźć harmonogram wykonywania szczegółowych zadań o minimalnym łącznym czasie realizacji. Narzędzia te są stosowane na wszystkich etapach realizacji przedsięwzięcia. W fazie strategicznej trudno jest jeszcze określić szczegółowe zadania, które będą musiały być realizowane, dlatego też często wykorzystuje się jedynie możliwości graficznego prezentowania harmonogramów, np. w formie wykresu Gantta (patrz rysunek 3.2).

Rozdział 4.
Faza określania wymagań

Celem tej fazy jest dokładne określenie wymagań klienta wobec tworzonego systemu. W fazie tej dokonywana jest więc zamiana celów klienta na konkretne wymagania zapewniające osiągnięcie tych celów. Należy pamiętać o tym, że klient rzadko dokładnie wie jakie wymagania zapewnią osiągnięcie jego celów. Proces określania wymagań należy więc rozumieć nie jako proste zbieranie wymagań klienta, lecz jako proces, w którym klient wspólnie z przedstawicielami producenta (analitykami) konstruuje zbiór wymagań wobec systemu zgodny z postawionymi celami.



Rysunek 4.1. Faza określania wymagań w cyklu życia oprogramowania

W przypadku firmy realizującej oprogramowanie na konkretne zamówienie, analitycy mają bezpośredni kontakt z przedstawicielami klienta. Podstawowym sposobem pracy jest więc wykonywanie wywiadów z różnymi osobami ze strony klienta. W odróżnieniu od fazy strategicznej, są to z reguły bezpośredni użytkownicy przyszłego systemu a nie zleceniodawcy. Faza ta wymaga dużego zaangażowania ze strony klienta. Najlepiej jest, jeżeli klient wyznacza osobę lub osoby, które będą odpowiedzialne za organizowanie współpracy analityków z przedstawicielami klienta. W wypadku dużych przedsięwzięć do tego celu wyznaczonych może zostać wiele osób, zawsze jednak klient powinien wyznaczyć konkretną osobę do pełnienia funkcji kierownika przedsięwzięcia ze strony klienta. Pomoc w realizacji fazy określania wymagań, jest dla przedstawicieli klienta dużym obciążeniem. Osoby te powinny być więc dodatkowo wynagradzane za wykonanie tej dodatkowej pracy. Wydatek taki jest niewielki w porównaniu z kosztem całego przedsięwzięcia i, biorąc pod uwagę duży koszt błędów popełnionych w tej fazie, jest z całą pewnością bardzo opłacalną inwestycją.

W przypadku firmy realizującej oprogramowanie sprzedawane rynkowo, korzystny jest również bezpośredni kontakt z osobami, które są przynajmniej potencjalnymi klientami i ekspertami z danej dziedziny. Faza określania wymagań wykonywana jest więc we współpracy z tymi osobami.

Trudność określania wymagań wynika z następujących czynników:

- Klient z reguły nie wie dokładanie w jaki sposób osiągnąć założone cele. Z drugiej strony cele klienta często można osiągnąć na wiele sposobów.
- Duże systemy są wykorzystywane przez wielu użytkowników. Ich cele są często sprzeczne. Różni użytkownicy mogą posługiwać się inną terminologią mówiąc o tych samych problemach.
- Zleceniodawcy i użytkownicy to często inne osoby. Głos zleceniodawców może być decydujący w tej fazie, choć nie zawsze potrafią oni właściwie przewidzieć potrzeby rzeczywistych użytkowników.

Wymagania klienta mogą być opisywane na różnych poziomach abstrakcji (ogólności):

- *Definicja wymagań* (ang. *requirement definition*), to ogólny opis w języku naturalnym. Opis taki jest wstępnym rezultatem rozmów z przedstawicielami klienta.
- *Specyfikacja wymagań* (ang. *requirements specification*), to częściowo ustrukturalizowany zapis wykorzystujący zarówno język naturalny, jaki i proste, częściowo przynajmniej sformalizowane notacje.
- *Specyfikacja oprogramowania* (ang. *software specification*), to w pełni formalny opis wymagań.

Specyfikacja oprogramowania jest rzadko wykonywana w praktyce. Często stosowanie formalnego opisu wymagań jest błędnie utożsamiane ze stosowaniem metody formalnych transformacji lub formalnymi dowodami poprawności. Formalna specyfikacja oprogramowania jest jednak także bardzo dobrą podstawą dla fazy testowania. Przykładem może być metoda Z schematów (ang. *Z schemas*) (Dilier, 1990).

Dobry opis wymagań powinien:

- być kompletny oraz niesprzeczny,
- opisywać zewnętrzne zachowanie systemu a nie sposób jego realizacji,
- obejmować ograniczenia przy jakich musi pracować system,
- być łatwy w modyfikacji,
- brać pod uwagę przyszłe możliwe zmiany wymagań wobec systemu,
- opisywać zachowanie systemu w niepożądanych sytuacjach.

Najbardziej chyba podstawowym błędem popełnianym podczas określania wymagań jest koncentrowanie się na sytuacjach typowych i pomijanie wyjątków. Zarówno użytkownicy, jak i analitycy mają tendencję do nie zauważania sytuacji nietypowych.

Wymagania wobec oprogramowania można podzielić na dwa zasadnicze rodzaje:

- *Wymagania funkcjonalne*. Opisują one funkcje (czynności, operacje) wykonywane przez system.
- *Wymagania niefunkcjonalne*. Opisują ograniczenia, przy zachowaniu których system powinien realizować swoje funkcje.

Wymagania powinny zostać zebrane w dokumencie — opisie wymagań. Dokument ten powinien być podstawą formalnego, szczegółowego kontraktu między klientem a producentem. Powinien też pozwalać na weryfikację czy stworzony system rzeczywiście spełnia postawione wymagania. Powinien więc być to dokument zrozumiały dla obu stron. Należy jednak zaznaczyć, że producenci oprogramowania często nie są zainteresowani precyzyjnym opisem wymagań, który pozwoliłby na rzeczywistą weryfikację wyprodukowanego systemu.

Dokument opisujący wymagania powinien obejmować następujące części:

- *Wprowadzenie* — cele, zakres i kontekst systemu. Ta część dokumentu zawiera więc wyniki fazy strategicznej (być może częściowo zmodyfikowane).
- *Opis ewolucji systemu* — opis przewidywanych zmian wymagań wobec systemu.
- *Opis wymagań funkcjonalnych*.
- *Opis wymagań niefunkcjonalnych*.
- *Model systemu*, którego tworzenie opisane jest dokładniej w kolejnym rozdziale.
- *Słownik*. Opis wymagań może zawierać szereg terminów niezrozumiałych dla jednej ze stron. Mogą to być zarówno terminy informatyczne niezrozumiałe dla przedstawicieli klienta, jak i terminy dotyczące dziedziny zastosowania systemu niezrozumiałe dla przedstawicieli producenta (w szczególności dla osób niezaangażowanych bezpośrednio w fazy określania wymagań i analizy). Słownik powinien wyjaśniać tego typu terminy. Słownik powinien także precyzować terminy niejednoznaczne, które używane są w dokumencie w znaczeniu, będącym tylko jednym z wielu znaczeń, jakie mogą pojawić się innych kontekstach.

Dokument ten może zostać uzupełniony o następujące dodatki:

- Specyfikacja wymagań funkcjonalnych.
- Specyfikacja wymagań niefunkcjonalnych.
- Wymagania sprzętowe.
- Wymagania dotyczące bazy danych.
- Indeks pomocny w wyszukiwaniu w dokumencie konkretnych informacji.

Tworząc dokument opisujący wymagania należy przestrzegać następujących zasad:

- wymagania funkcjonalne powinny być oddzielone od wymagań niefunkcjonalnych,
- należy rozdzielać opisy poszczególnych wymagań, np. umieszczając opis każdego z nich w odrębnym akapicie,
- dla każdego wymagania powinien być podawany powód jego wprowadzenia — cel lub cele przedsięwzięcia, których osiągnięcie wspomaga dane wymaganie.

Określone w tej fazie wymagania będą podstawą testowania zrealizowanego systemu. W tej fazie powinien więc powstać także plan testów.

4.1. Wymagania funkcjonalne

Wymagania funkcjonalne opisują funkcje (czynności, operacje) wykonywane przez system. Funkcje te mogą być wykonywane przy udziale systemów zewnętrznych. Na przykład funkcja „Ewidencja podatników" realizowana przez system podatkowy, jest wykonywana przy udziale użytkownika, który wprowadza, usuwa i modyfikuje informacje o podatnikach.

Jak wspomniano powyżej język naturalny jest często stosowanym sposobem opisu wymagań. Stosowanie języka naturalnego ma jednak następujące wady:

- Niejednoznaczność języka naturalnego, która utrudnia precyzyjny zapis wymagań. Może ona także prowadzić do różnej interpretacji tego samego tekstu przez różne osoby.
- Elastyczność języka naturalnego, która pozwala te same treści wyrazić na różne sposoby. Utrudnia to wykrycie powiązanych wymagań. Może też prowadzić do przeoczenia sprzeczności wymagań sformułowanych w różny sposób lecz dotyczących tych samych funkcji.

Powyższych wad pozbawione są notacje formalne, które jednak są w zdecydowanej większości przypadków niezrozumiałe dla klienta. Mogą one więc być jedynie uzupełnieniem zapisu słownego.

Swego rodzaju kompromisem pomiędzy powyższymi metodami jest stosowanie formularzy do zapisu wymagań funkcjonalnych. Formularze takie wykorzystują język naturalny. Zapis jest jednak podzielony na konkretne pola, co pozwala uniknąć szeregu błędów w określaniu wymagań. Poniżej podany jest przykład formularza opisującego wymaganie funkcjonalne (tabela 4.1).

Definiując wymagania funkcjonalne z reguły podaje się jedynie wymagany efekt realizacji funkcji a nie jej algorytm. Jest więc to deklaratywny opis systemu. W szczególnych wypadkach wymagane może być jednak zastosowanie konkretnego algorytmu. Może można uznać za wymaganie niefunkcjonalne związane z realizacją danej funkcji.

Tabela 4.1.
Przykład formularza opisu wymagania

Nazwa funkcji	*Edycja dochodów podatnika*
Opis	Funkcja pozwala edytować łączne dochody podatnika uzyskane w danym roku.
Dane wejściowe	Informacje o dochodach podatnika uzyskanych z różnych źródeł: kwoty przychodów, kosztów uzyskania przychodów oraz zapłaconych zaliczek na poczet podatku dochodowego. Informacje o dokumentach opisujących dochody z poszczególnych źródeł.
Źródło danych wejściowych	Dokumenty oraz informacje dostarczone przez podatnika. Dane wpisywane przez pracownika firmy podatkowej.
Wynik	
Warunek wstępny	Kwota dochodu = kwota przychodu – kwota kosztów (zarówno dla konkretnych dochodów, jaki i dla łącznych dochodów podatnika). Łączne kwoty przychodów, kosztów uzyskania, dochodów oraz zapłaconych zaliczek są sumami tych kwot dla dochodów z poszczególnych źródeł.
Warunek końcowy	Kwota dochodu = kwota przychodu – kwota kosztów (zarówno dla konkretnych dochodów, jaki i dla łącznych dochodów podatnika). Łączne kwoty przychodów, kosztów uzyskania, dochodów oraz zapłaconych zaliczek są sumami tych kwot dla dochodów z poszczególnych źródeł.
Efekty uboczne	Uaktualnienie podstawy opodatkowania.
Powód	Funkcja pomaga przyspieszyć obsługę klientów oraz zmniejszyć ryzyko popełnienia błędów.

Liczba wymagań funkcjonalnych może być bardzo duża. W literaturze można znaleźć informacje o systemach, w których wyróżniono nawet 8 000 różnych wymagań funkcjonalnych (Bell i in., 1977). Konieczne jest pewnego rodzaju uporządkowanie tych wymagań, które ułatwi pracę nad nimi. Poniżej omawiane są dwie, niewykluczające się zresztą metody, ułatwiające zapanowanie nad dużą liczbą wymagań. Są to: hierarchiczny zapis wymagań oraz diagramy przypadków użycia.

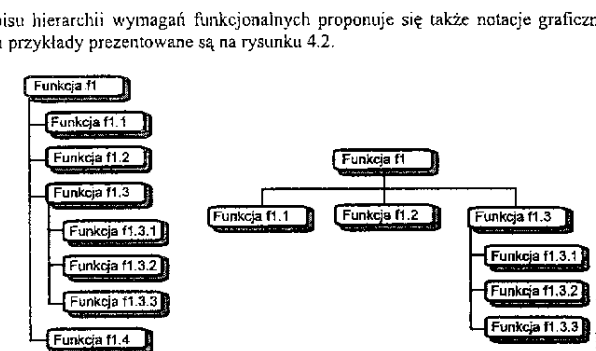
4.1.1. Hierarchia wymagań funkcjonalnych

Metoda ta opiera się na obserwacji faktu, że pewne funkcje wykonywane przez system są składowymi (podfunkcjami) innych bardziej ogólnych funkcji. Funkcja polegająca na projektowaniu systemu informacji geograficznej składa się z takich funkcji, jak: edycja dla, edycja obiektów umieszczonych na mapie, edycja słów kluczowych. Proponuje się więc, aby wymagania funkcjonalne zapisywać w formie hierarchicznej. Hierarchia taka może być opisana w formie tekstowej w sposób podany poniżej.

Tekstowy zapis hierarchii wymagań funkcjonalnych

Funkcja nadrzędna f1
- funkcja f1.1
- funkcja f1.2
- funkcja f1.3
  - funkcja f1.3.1
  - funkcja f1.3.2
  - funkcja f1.3.3
- funkcja f1.4
  - funkcja f1.4.1
  - funkcja f1.4.2
  - funkcja f1.4.3

Do zapisu hierarchii wymagań funkcjonalnych proponuje się także notacje graficzne, których przykłady prezentowane są na rysunku 4.2.



Rysunek 4.2. Przykłady graficznego zapisu hierarchii wymagań funkcjonalnych

Dana funkcja powinna być dekomponowana na wszystkie, i wyłącznie te funkcje, któ są niezbędne dla jej wykonania. Kolejność funkcji składowych nie ma znaczenia. W konanie funkcji nadrzędnej nie musi w każdej sytuacji oznaczać wykonania wszystki funkcji składowych. Z drugiej strony pewne funkcje składowe mogą być wykonywa wielokrotnie podczas realizacji funkcji podrzędnej. Każda z funkcji składowych m być jednak w pewnych sytuacjach wykonywana podczas realizacji funkcji nadrzędne

Wymagania funkcjonalne powinny dotyczyć wyłącznie ze wnętrznych funkcji syste Z reguły oznacza to, że funkcje te powinny być widoczne dla użytkownika. Jeżeli

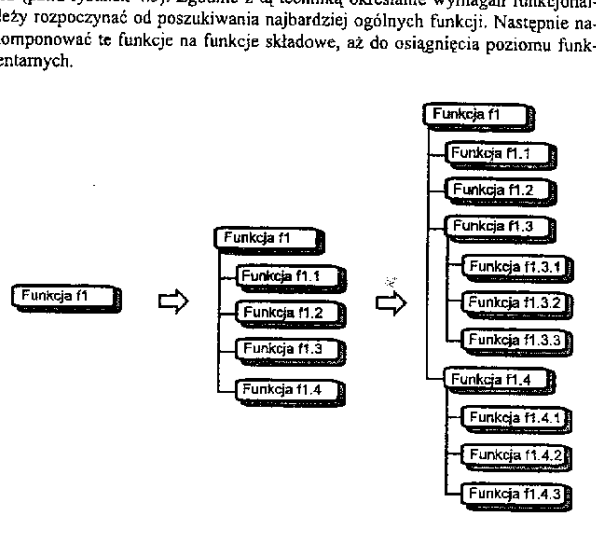
tem współpracuje z innymi systemami zewnętrznymi, może to także oznaczać widoczność tych funkcji np. dla innych programów. Funkcji systemu nie należy rozbijać do poziomu funkcji, które mają znaczenie czysto implementacyjne, nawet jeżeli poszczególne wymagania funkcjonalne są bardzo złożone i na etapie implementacji będą realizowane przez szereg składowych oprogramowania.

Pewne funkcje składowe mogą pojawiać się w ramach wielu funkcji nadrzędnych. Sytuacje takie powinny być wyróżniane w szczególny sposób. Przykład podany jest poniżej.

Kopie funkcji

Funkcja nadrzędna f1
- funkcja f1.1
  - funkcja f1.1.1
  - funkcja f1.1.2
- funkcja f1.2
  - funkcja f1.2.1
  - funkcja f1.2.2 (kopia funkcji f1.1.2)

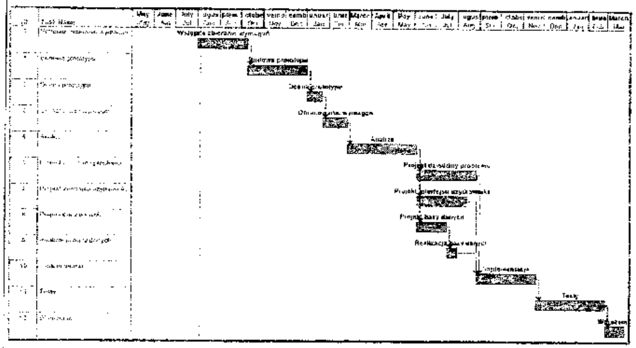
Jednym z proponowanych sposobów konstruowania hierarchii funkcji jest technika zstępująca (patrz rysunek 4.3). Zgodnie z tą techniką określanie wymagań funkcjonalnych należy rozpoczynać od poszukiwania najbardziej ogólnych funkcji. Następnie należy dekomponować te funkcje na funkcje składowe, aż do osiągnięcia poziomu funkcji elementarnych.



Rysunek 4.3. Zstępujące konstruowanie hierarchii wymagań

W przypadku firmy realizującej oprogramowanie sprzedawane rynkowo, zgromadzone wyniki są prezentowane kierownictwu wyższego szczebla, które podejmuje decyzję o rozpoczęciu lub zaniechaniu produkcji oprogramowania.

W fazie strategicznej określany jest także wstępny harmonogram przedsięwzięcia. Czynność ta polega na podziale przedsięwzięcia na pewne mniejsze zadania, określeniu terminów ich realizacji oraz zasobów niezbędnych do ich wykonania. Techniki harmonogramowania przedsięwzięć omawiane są w sekcji 13.8. Na tym etapie nie jest jeszcze możliwe wyróżnienie szczegółowych zadań. Harmonogram jest bardzo ogólny i musi być uszczegóławiany w trakcie realizacji przedsięwzięcia. Wygodnym sposobem prezentowania harmonogramów są tak zwane wykresy Gantta. Przykład takiego wykresu opracowanego za pomocą pakietu MS Project podany jest na rysunku 3.2.



Rysunek 3.2. Przykład wstępnego harmonogramu [illegible]

W fazie strategicznej należy także zdefiniować standardy określające wymagany sposób pracy w dalszych fazach przedsięwzięcia. Standardy te powinny opisywać:

- wykorzystanie konkretnych narzędzi i notacji,
- sposób opracowywania dokumentacji.

Dobrze zorganizowana firma powinna w dużym stopniu wykorzystywać te same standardy w różnych przedsięwzięciach. Może jednak zachodzić konieczność ich dostosowania do potrzeb konkretnego zadania. Zmiana standardów jest oczywiście konieczna, jeżeli nastąpi istotna zmiana sposobu pracy, np. stosowane są nowe techniki i/lub narzędzia. Dalsze zagadnienia dotyczące standardów omawiane są w rozdziale 13.3 omawiającym zapewnianie jakości.

3.1. Ocena rozwiązań

Jak wspomniano powyżej, wybór najlepszego sposobu realizacji przedsięwzięcia (rozwiązania) jest podstawowym warunkiem końcowego sukcesu. W fazie strategicznej rozważanych jest często kilka możliwych rozwiązań, z których następnie wybiera się jedno najlepsze. Istnieją dwa podstawowe źródła trudności w porównywaniu rozwiązań:

- wielość celów przedsięwzięcia (z punktu widzenia producenta), czyli wielość kryteriów oceny proponowanych rozwiązań,
- niepewność, tj. fakt niemożliwości precyzyjnej oceny spodziewanych rezultatów wyboru danego rozwiązania.

Kryteria stosowane do oceny rozwiązań mogą być różne w zależności od kontekstu danego przedsięwzięcia. Firma może kierować się np. takimi kryteriami jak:

- koszt,
- czas realizacji,
- niezawodność,
- stopień możliwości ponownego wykorzystania fragmentów systemu,
- przenośność na inne platformy,
- wydajność.

Wygodnym sposobem prezentowania poszczególnych rozwiązań jest zapis tabelaryczny, w którym wszystkie rozwiązania opisane są wartościami poszczególnych kryteriów. Przykładem może być tabela 3.1 zawierająca opis trzech rozwiązań.

Tabela 3.1.
Tabelaryczny zapis rozważanych rozwiązań

Rozwiązanie	*A*	*B*	*C*
Koszt [tys. zł]	120	80	175
Czas [miesiące]	33	30	36
Niezawodność [liczba błędnych wykonań/tydzień]	5	9	13
Ponowne wykorzystanie [%]	40	40	30
Przenośność [%]	90	75	30
Wydajność [transakcji/s]	0,35	0,75	1

Pierwszym krokiem powinno być usunięcie rozwiązań zdominowanych. Rozwiązanie jest zdominowane jeżeli istnieje inne rozwiązanie nie gorsze z punktu widzenia żadnego kryterium i lepsze na co najmniej jednym kryterium. Gdyby przewidywana wydajność dla rozwiązania C wynosiła 0,75 [transakcji/s] byłoby to rozwiązanie zdominowane, gdyż rozwiązanie B daje taką samą wydajność i jest lepsze pod względem innych kryteriów. Jeżeli różnica pomiędzy wydajnością 0,75 [transakcji/s] a 1 [transakcji/s] jest zdaniem kierownictwa firmy nieistotna, można rozwiązanie C uznać za „praktycznie zdominowane" i wyeliminować z dalszych rozważań. W praktyce najczęściej poszczególne rozwiązania są szczególnie dobre z punktu widzenia pewnych kryteriów, rzadko więc pojawiają się rozwiązania zdominowane.

Poszczególne kryteria oceny różnią się z reguły swoim znaczeniem. Można im więc nadać pewne wagi, które oczywiście mogą być różne w różnych przedsięwzięciach. Wagi pozwalają obliczyć łączną ocenę poszczególnych rozwiązań, będącą sumą ważoną wszystkich kryteriów. Ponieważ jednak poszczególne kryteria wyrażane są w różnych jednostkach i przyjmują wartości z zupełnie różnych zakresów nie jest sensowne dodawanie do siebie wartości tych kryteriów nawet przy uwzględnieniu wag. Konieczne jest więc znormalizowanie wartości kryteriów, tak aby wszystkie przyjmowały wartości z podobnego zakresu. Wszystkie kryteria można np. znormalizować do zakresu [0,1] gdzie 0 oznacza najgorszą wartość a 1 wartość najlepszą. Wartość 0 można przydzielić najgorszej rzeczywistej wartości danego kryterium, wartość 1 najlepszej rzeczywistej wartości, a pozostałym wartościom kryteriów proporcjonalnie obliczone wartości pośrednie. Wartości znormalizowane, po przemnożeniu przez wagi kryteriów, mogą być zsumowane w celu otrzymania łącznej oceny rozwiązań. Tabela 3.2 zawiera znormalizowane wartości kryteriów dla sytuacji opisanej w tabeli 3.1, wagi poszczególnych kryteriów i łączne oceny rozwiązań. Jak widać, w tym wypadku rozwiązanie B może być uznane za najlepsze. Arkusze kalkulacyjne są wygodnymi narzędziami do wykonania tego typu obliczeń. Pozwalają one na szybką analizę wpływu zmian wartości wag lub zmian wartości pewnych kryteriów na łączną ocenę.

Suma ważona jest tylko jednym z wielu możliwych sposobów oceny opcji w obecności wielu kryteriów. Obszerna dziedzina zajmująca się tymi zagadnieniami nosi nazwę wielokryterialnego wspomagania decyzji (Roy, 1985).

Tabela 3.2.
Łączna ocena rozwiązań za pomocą sumy ważonej

Rozwiązanie	*A*	*B*	*C*	*Waga*
Koszt	0,58	1	0	3
Czas	0,5	1	0	2
Niezawodność	1	0,5	0	3
Ponowne wykorzystanie	1	1	0	1
Przenośność	1	0,75	0	1
Wydajność	0	0,62	1	1,5
Łączna ocena	**7,74**	**9,17**	**1,5**	

Niepewność jest kolejnym czynnikiem utrudniającym wybór najlepszego rozwiązania. Przykładem może być sytuacja firmy, która rozważa wykorzystanie nowego środowiska programistycznego (rozwiązanie A) lub zastosowanie dotychczasowego (rozwiązanie B). Dla każdego z tych rozwiązań oszacowano koszt optymistyczny i pesymistyczny (tabela 3.3). Jak widać rozrzut tych kosztów dla opcji B jest znacznie mniejszy.

Tabela 3.3.
Przykład niepewności w ocenie kosztu

Rozwiązanie	*A*	*B*
Pesymistyczny koszt [tys zł]	100	80
Optymistyczny koszt [tys zł]	40	65

Jeżeli nie są dostępne żadne dodatkowe informacje, osoby oceniające te rozwiązania mogą wybrać jedną z dwóch strategii:

- strategię konserwatywną (pesymistyczną), która bierze pod uwagę szacunki pesymistyczne,
- strategię optymistyczną, która bierze pod uwagę szacunki optymistyczne.

W pierwszym wypadku zostałoby wybrane rozwiązanie B, w drugim rozwiązanie A. Nie można w obiektywny sposób stwierdzić, która z tych strategii jest lepsza.

Bardziej racjonalna ocena jest możliwa jeżeli dodatkowo zostaną oszacowane tak zwane prawdopodobieństwa subiektywne zajścia poszczególnych zdarzeń pod warunkiem wybrania danego rozwiązania. W tym celu należy oszacować prawdopodobieństwa sukcesu i porażki (osiągnięcia optymistycznego i pesymistycznego kosztu) przy wyborze rozwiązań A i B. Przykładowe prawdopodobieństwa podane są w tabeli 3.4.

Tabela 3.4.
Prawdopodobieństwa subiektywne

Rozwiązanie	*A*	*B*
Prawdopodobieństwo pesymistycznego rozwiązania	0,5	0,2
Prawdopodobieństwo optymistycznego rozwiązania	0,5	0,8

Powyższe informacje pozwalają obliczyć tak zwany spodziewany koszt dla poszczególnych opcji:

Spodziewany koszt rozwiązania A = koszt pesymistyczny * prawdopodobieństwo zajścia tego zdarzenia + koszt optymistyczny * prawdopodobieństwo zajścia tego zdarzenia = = 0,5 * 100 + 0,5 * 40 = 70 [tys. zł].

Spodziewany koszt rozwiązania B = 0,2 * 80 + 0,8 * 65 = 68 [tys. zł]

Spodziewany koszt jest więc niższy dla rozwiązania B. Jeżeli jednak prawdopodobieństwo sukcesu w przypadku rozwiązania A oceniono by na 0,4 uzyskano by następujący wynik:

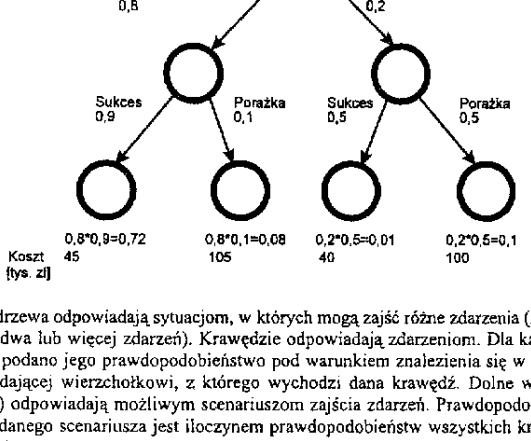
Spodziewany koszt rozwiązania A = 0,4 * 100 + 0,6 * 40 = 64 [tys. zł]

Spodziewany koszt rozwiązania B = 0,2 * 80 + 0,8 * 65 = 68 [tys. zł]

Racjonalny byłby wówczas wybór rozwiązania A.

W wielu sytuacjach należy brać pod uwagę wiele zdarzeń, które mogą zajść w wypadku wyboru pewnego rozwiązania. Firma pragnąca zastosować nowe środowisko programistyczne może na przykład rozważać zatrudnienie zewnętrznego eksperta na czas realizacji przedsięwzięcia. Nie jest jednak pewne czy eksperta uda się znaleźć. Prawdopodobieństwo znalezienia eksperta oceniono na 0,8. Jeżeli ekspert zostanie znaleziony, szansę osiągnięcia optymistycznego kosztu znacznie rosną i wynoszą 0,9. Do analizy tego typu sytuacji pomocne są tzw. drzewa ryzyka. Przykład drzewa ryzyka zaprezentowany jest na rysunku 3.3.

Rysunek 3.3. Przykład drzewa ryzyka



Wierzchołki drzewa odpowiadają sytuacjom, w których mogą zajść różne zdarzenia (może na rozważać dwa lub więcej zdarzeń). Krawędzie odpowiadają zdarzeniom. Dla każdego zdarzenia podano jego prawdopodobieństwo pod warunkiem znalezienia się w sytuacji odpowiadającej wierzchołkowi, z którego wychodzi dana krawędź. Dolne wierzchołki (liście) odpowiadają możliwym scenariuszom zajścia zdarzeń. Prawdopodobieństwo zajścia danego scenariusza jest iloczynem prawdopodobieństw wszystkich krawędzi jakie do niego prowadzą. Informacje zaprezentowane na rysunku 3.3 pozwalają obliczyć spodziewany koszt rozwiązania polegającego na zastosowaniu nowego środowiska programistycznego i próbie zatrudnienia zewnętrznego eksperta:

Spodziewany koszt = 45 * 0,72 + 105 * 0,08 + 40 * 0,5 + 100 * 0,5 = 54,8 [tys. zł]

3.2. Szacowanie kosztów oprogramowania

Wstępne oszacowanie kosztów oprogramowania musi zostać wykonane dla każdego z rozważanych rozwiązań. Dla ostatecznie wybranego rozwiązania niezbędne jest wykonanie dokładniejszego oszacowania.

Na koszt oprogramowania składają się następujące główne czynniki:

- koszt sprzętu będącego częścią tworzonego systemu,
- koszt wyjazdów i szkoleń,
- koszt zakupu narzędzi,
- nakład pracy.

Trzy pierwsze czynniki są stosunkowo łatwe do oszacowania, natomiast ocena nakładów pracy niezbędnych dla zrealizowania systemu jest bardzo trudna. Dlatego też szacowanie kosztów oprogramowania jest praktycznie utożsamiane z szacowaniem nakładów pracy.

Dla oszacowania kosztów oprogramowania (nakładów pracy) stosuje się następujące techniki:

- *Modele algorytmiczne.* Techniki te wymagają opisu danego przedsięwzięcia za pomocą szeregu atrybutów liczbowych i/lub opisowych. Odpowiedni algorytm (często zwykła formuła matematyczna) daje następnie w wyniku spodziewany nakład pracy.
- *Ocena przez eksperta(ów).* Doświadczone osoby często z dużą precyzją potrafią oszacować koszt realizacji nowego systemu.
- *Ocena przez analogię.* Metoda ta wymaga dostępu do informacji o poprzednio realizowanych przedsięwzięciach. Mogą to być przedsięwzięcia wykonywane w firmie, która wykonuje szacowanie kosztów lub w innych firmach o podobnym profilu. Komercyjnie sprzedawane są np. bazy danych zawierających informacje o różnorodnych przedsięwzięciach realizowanych przez rozmaite firmy. Ogólnie rzecz biorąc metoda ta polega na wyszukaniu przedsięwzięcia (przedsięwzięć) podobnych do aktualnie rozważanego i wykorzystaniu informacji o nakładach poniesionych na ich realizację. Do tej grupy technik należy też zaliczyć stosowanie systemów automatycznego uczenia się (sieci neuronowych, systemów regułowych). System taki uczy się na podstawie pewnego zbioru przedsięwzięć, a następnie jest wykorzystywany do oceny nowych przedsięwzięć.
- *Prawo Parkinsona.* Jest to prawo, które stwierdza, że przedsięwzięcia, w tym także przedsięwzięcia programistyczne, praktycznie zawsze są wykonywane przy założonych nakładach.

- *Wycena dla wygranej* (ang. *pricing to win*). Koszt oprogramowania szacowany jest na podstawie oceny możliwości klienta(ów) oraz przewidywanych działań konkurentów. Prawo Parkinsona pozwala wierzyć, że przedsięwzięcie i tak zmieści się w założonych nakładach. Moje doświadczenie wskazuje, że jest to technika często stosowana przez firmy biorące udział w przetargach na opracowanie oprogramowania.

- *Szacowanie wstępujące*. Realizację przedsięwzięcia dzieli się mniejsze zadania, których koszt jest łatwiejszy do oszacowania. Następnie oblicza się koszt całego przedsięwzięcia. Przykładem szacowania wstępującego są techniki harmonogramowania omawiane w sekcji 13.8.

Pewnego komentarza wymaga prawo Parkinsona. Jaki jest mechanizm powodujący, że przedsięwzięcia programistyczne mają tendencję do dokładnego mieszczenia się w założonych nakładach? Moje własne doświadczenie oraz konsultacje z kierownikami firm programistycznych wskazują na następujący mechanizm. Jeżeli nakład pracy zostanie przeszacowany, dodatkowy czas zostanie bez trudu zagospodarowany przez twórców oprogramowania na czynności w niewielkim stopniu wpływające na rzeczywistą jakość końcowego produktu. Kierownik pewnego przedsięwzięcia stwierdził, że „jeżeli programista wykona w ciągu jednego dnia zadanie, na które dostał tydzień, to potrafi poświęcić pozostałe cztery dni na to, by na karcie graficznej z ośmioma kolorami uzyskać dziewiąty kolor". Jeżeli natomiast nakład pracy zostanie niedoszacowany, kierownictwo przedsięwzięcia może dążyć do zmieszczenia się planach poprzez rezygnację z części funkcji systemu i/lub przez obniżenie niezawodności oprogramowania (na przykład redukując czas trwania testów). Obniża to jednak w oczywisty sposób jakość systemu.

3.3. Algorytmiczne modele szacowania kosztów oprogramowania — model COCOMO

Stosowanie modeli algorytmicznych rozpoczyna się od opisania danego przedsięwzięcia za pomocą szeregu atrybutów. W zdecydowanej większości przypadków jednym z tych atrybutów jest rozmiar systemu, mierzony np. liczbą instrukcji kodu źródłowego. Modele takie opierają się więc na założeniu, że oszacowanie rozmiaru systemu jest znacznie łatwiejsze niż oszacowania nakładu pracy.

Proponuje się oczywiście także modele algorytmiczne, które nie opierają się na oszacowaniu liczby instrukcji kodu. Przykładem może być metoda punktów funkcyjnych (ang. *function points*). Metoda ta opiera się na oszacowaniu następujących czynników wpływających na złożoność systemu:

- dane odczytywane przez system i pobierane z systemu,

- interakcja z użytkownikiem,

- zewnętrzne interfejsy,

- pliki wykorzystywane przez system.

Jednym z częściej wykorzystywanych modeli algorytmicznych jest model COCOMO (COst COnstruction MOdel). Model ten powstał w oparciu o informacje dotyczące wielu rzeczywistych przedsięwzięć. Ponieważ jednak przedsięwzięcia te były realizowane w sposób tradycyjny, bez wykorzystania narzędzi CASE, można się spodziewać, że model ten nie jest w pełni adekwatny w przypadku współczesnych przedsięwzięć. Dlatego też nie są omawiane wszystkie szczegóły tego modelu. Z drugiej strony, znajomość czynników branych pod uwagę w tym modelu jest niewątpliwie przydatna, nawet jeżeli konkretne parametry liczbowe tego modelu nie w pełni odpowiadają przedsięwzięciom wykorzystującym narzędzia CASE. Interesująca jest także ogólna postać zależności pojawiających się w tym modelu.

Stosowanie modelu COCOMO wymaga oszacowania liczby instrukcji, z których będzie się składał system. Rozważane przedsięwzięcie jest następnie zaliczane do jednej z klas:

- *Przedsięwzięć organicznych* (ang. *organic projects*). Klasa ta obejmuje przedsięwzięcia wykonywane przez stosunkowo małe zespoły, złożone z osób o podobnym, wysokim poziomie umiejętności technicznych. Dziedzina problemu jest dość dobrze znana. Przedsięwzięcie jest wykonywane za pomocą dobrze znanych metod i narzędzi.

- *Przedsięwzięć półoderwanych* (ang. *semi-detached projects*). Członkowie zespołu różnią się stopniem zaawansowania. Pewne aspekty dziedziny problemu, część metod i narzędzi nie jest dobrze znana.

- *Przedsięwzięć osadzonych* (ang. *embeded projects*). Klasa ta obejmuje przedsięwzięcia polegające na realizacji systemów o bardzo złożonych wymaganiach (np. wymagających ścisłej współpracy ze sprzętem). Dziedzina problemu, stosowane metody i narzędzia są w dużej mierze nieznane. Większość członków zespołu nie ma doświadczenia w realizacji podobnych zadań.

Podstawowe wyrażenie stosowane w tym modelu pozwalające wyznaczyć nakład pracy to:

$$\text{Nakład [osobomiesiące]} = A\,(\text{KDSI})^b,$$

gdzie KDSI oznacza tysiąc instrukcji kodu źródłowego (ang. thousand of delivered source code instructions). KDSI nie obejmuje więc fragmentów kodu, które nie zostały ostatecznie wykorzystane w końcowym systemie, mimo że ich opracowanie mogło wymagać sporych nakładów.

Wartości stałych A i b zależą od klasy, do której zaliczono przedsięwzięcie:

- przedsięwzięcia organiczne:

$$\text{Nakład} = 2{,}4\,(\text{KDSI})^{1{,}05}$$

- przedsięwzięcia półoderwane:

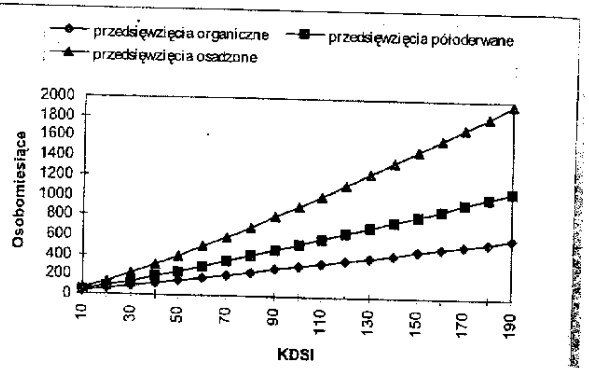
  Nakład = 3 (KDSI)^{1,12}

- przedsięwzięcia osadzone:

  Nakład = 3,6 (KDSI)^{1,20}

Rysunek 3.4. w sposób graficzny obrazuje zależność nakładów od rozmiaru systemu dl poszczególnych klas przedsięwzięć. Zależność ta jest nieliniowa, nakłady rosną wię szybciej niż proporcjonalnie do liczby instrukcji kodu źródłowego. Wzrost ten jes szczególnie szybki dla trudnych przedsięwzięć. Dla niewielkich, stosunkowo prostyc przedsięwzięć zależność ta jest bliska liniowej.

Rysunek 3.4. Oszacowania nakładów w modelu COCOMO



Model COCOMO zakłada także, że znając nakład pracy można oszacować czas rea zacji przedsięwzięcia, z czego wynika oczywiście przybliżona wielkość zespołu, któ powinien realizować przedsięwzięcie. Wynika to z obserwacji, że dla każdego przed sięwzięcia istnieje optymalna wielkość zespołu wykonawców. Zwiększenie liczby osó może nawet spowodować wydłużenie czasu realizacji. Proponowane wyrażenia, pozwa lające oszacować czas trwania przedsięwzięć, są następujące:

- przedsięwzięcia organiczne:

  Czas [miesiące] = 2,5 (Nakład)^{0,32},

- przedsięwzięcia półoderwane:

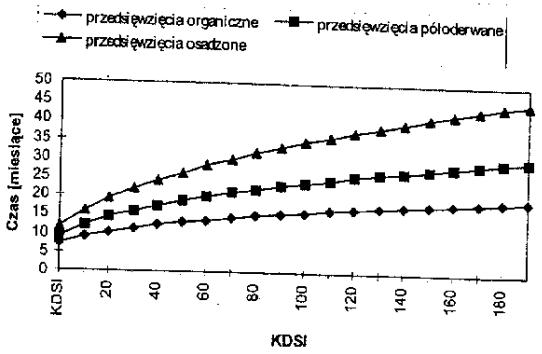
  Czas [miesiące] = 2,5 (Nakład)^{0,35},

- przedsięwzięcia osadzone:

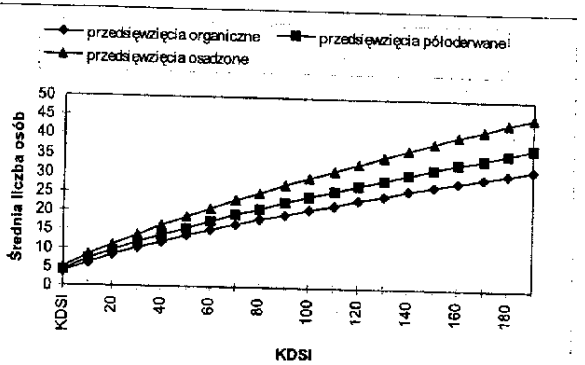
  Czas [miesiące] = 2,5 (Nakład)^{0,38}.

Zależności te są w sposób graficzny zobrazowane na rysunkach 3.5 i 3.6.

Rysunek 3.5. Oszacowania czasu trwania przedsięwzięć w modelu COCOMO



Rysunek 3.6. Oszacowania średniej liczby osób niezbędnych do realizacji przedsięwzięcia w modelu COCOMO

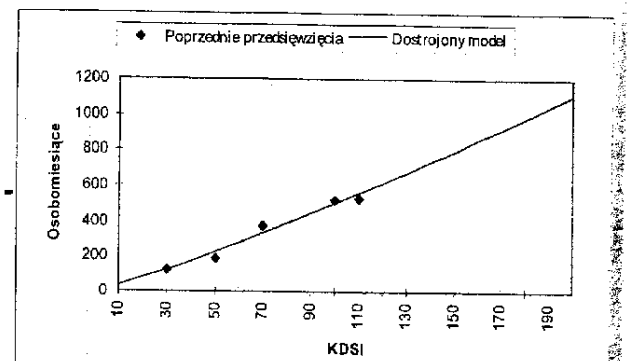


Przedstawione powyżej zależności to podstawowy model COCOMO. Otrzymane w ten sposób oszacowania powinny zostać skorygowane za pomocą tzw. czynników modyfikujących. Opierają się one na oszacowaniu pewnych dodatkowych atrybutów opisujących przedsięwzięcie. Z każdym atrybutem wiąże się pewien czynnik modyfikujący nieznacznie mniejszy lub większy od jedności. Oszacowane za pomocą modelu podstawowego nakłady mnoży się następnie przez czynniki modyfikujące. Wartość czynnika większa od jedności wpływa więc na wzrost nakładów, a wartość mniejsza od jedności na ich zmniejszenie. Czynniki modyfikujące tworzy się biorąc pod uwagę następujące atrybuty przedsięwzięcia:

- wymagania wobec niezawodności systemu,

- rozmiar bazy danych w stosunku do rozmiaru kodu,
- złożoność systemu, tj. złożoność struktur danych, złożoność algorytmów, komunikacja z innymi systemami, stosowanie obliczeń równoległych,
- wymagania co do wydajności systemu,
- ograniczenia pamięci,
- zmienność maszyny wirtualnej, tj. zmienność sprzętu i oprogramowania systemowego tworzących środowisko pracy systemu.

Liczne obserwacje wskazują na bardzo znaczące różnice w wydajności poszczególnych firm programistycznych. Podobne przedsięwzięcia mogą być realizowane przez pewne firmy przy dziesięciokrotnie niższych nakładach. Powstaje więc pytanie, czy przy tak znacznym rozrzucie wydajności możliwe jest sensowne stosowanie modelu COCOMO i innych modeli algorytmicznych? Czy nie są to modele opisujące wyłącznie jakieś wyidealizowane średnie przedsiębiorstwo? Praktyczne wykorzystanie modeli algorytmicznych wymaga ich dostrojenia (kalibracji) do sytuacji danej firmy. Do tego celu niezbędne są dane liczbowe dotyczące poprzednich przedsięwzięć. Na ich podstawie można — stosując na przykład metodę najmniejszych kwadratów — znaleźć stałe A i b, dające przewidywania maksymalnie zgodne z obserwacjami (patrz rysunek 3.7).



Rysunek 3.7. Dostrajanie modelu algorytmicznego na podstawie danych o poprzednich przedsięwzięciach

Wykonywanie tego typu czynności wspomagają typowe arkusze kalkulacyjne i programy statystyczne. Skalibrowany w ten sposób model algorytmiczny może być bardzo precyzyjnym narzędziem szacowania kosztów nowych systemów, szczególnie jeżeli nowe przedsięwzięcia są podobne do realizowanych poprzednio.

3.4. Podsumowanie

3.4.1. Kluczowe czynniki sukcesu

Zakończona sukcesem realizacja fazy strategicznej zależy oczywiście od wielu czynników. Poniżej wymieniono najistotniejsze z nich, czyli tzw. kluczowe czynniki sukcesu:

- *Szybkość pracy.* Szczególnie w przypadku firm realizujących oprogramowanie na zamówienie, startujących w przetargach, opóźnienia w realizacji tej fazy mogą zaprzepaścić szansę na uzyskanie zamówienia. Niezależnie od rodzaju tworzonego systemu, firma programistyczna z reguły nie może sobie pozwolić na przeznaczenie zbyt dużych nakładów na tę fazę. Wymaga to więc wykonania tej fazy przez stosunkowo niewielką liczbę osób w krótkim czasie.
- *Zaangażowanie kluczowych osób ze strony klienta.* Brak akceptacji dla sposobu realizacji przedsięwzięcia ze strony pewnych kluczowych osób może praktycznie uniemożliwić jego przyszły sukces.
- *Uchwycenie (ogólne) całości systemu.* Podstawowym błędem popełnianym w tej fazie jest zbyt szybka koncentracja na pewnych fragmentach systemu. Niemożliwe jest w takiej sytuacji właściwe oszacowanie kosztów wykonania systemu. Łatwo jest także przeoczyć szczególnie trudne fragmenty systemu.

3.4.2. Podstawowe rezultaty fazy strategicznej

Podstawowe rezultaty fazy strategicznej to:

- udostępniany klientowi raport obejmujący:
  - definicję celów przedsięwzięcia,
  - opis zakresu przedsięwzięcia,
  - opis systemów zewnętrznych, z którymi system będzie współpracować,
  - ogólny opis wymagań,
  - ogólny model systemu,
  - opis proponowanego rozwiązania,
  - oszacowanie kosztów,
  - wstępny harmonogram prac,
- raport oceny rozwiązań, zawierający informacje o rozważanych rozwiązaniach oraz przyczynach wyboru jednego z nich,
- opis wymaganych zasobów — pracownicy, oprogramowanie, sprzęt,
- definicje standardów,
- harmonogram fazy analizy.

3.4.3. Narzędzia CASE w fazie strategicznej

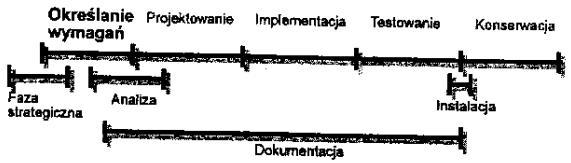
Ponieważ faza strategiczna obejmuje ogólne określenie wymagań, wstępną analizę oraz projekt, wykorzystywane są w tej fazie te same narzędzia, które będą wykorzystywane w kolejnych fazach. Narzędzia te zostaną omówione dokładniej przy szczegółowym omawianiu tych faz.

Na rynku dostępne są programy wspomagające stosowanie rozmaitych metod szacowania kosztów oprogramowania. Mogą one być częściami większych systemów CASE lub niezależnymi programami.

W fazie strategicznej wykorzystuje się także narzędzia harmonogramowania. Pozwalają one np. znaleźć harmonogram wykonywania szczegółowych zadań o minimalnym łącznym czasie realizacji. Narzędzia te są stosowane na wszystkich etapach realizacji przedsięwzięcia. W fazie strategicznej trudno jest jeszcze określić szczegółowe zadania, które będą musiały być realizowane, dlatego też często wykorzystuje się jedynie możliwości graficznego prezentowania harmonogramów, np. w formie wykresu Gantta (patrz rysunek 3.2).

Rozdział 4.
Faza określania wymagań

Celem tej fazy jest dokładne określenie wymagań klienta wobec tworzonego systemu. W fazie tej dokonywana jest więc zamiana celów klienta na konkretne wymagania zapewniające osiągnięcie tych celów. Należy pamiętać o tym, że klient rzadko dokładnie wie jakie wymagania zapewnią osiągnięcie jego celów. Proces określania wymagań należy więc rozumieć nie jako proste zbieranie wymagań klienta, lecz jako proces, w którym klient wspólnie z przedstawicielami producenta (analitykami) konstruuje zbiór wymagań wobec systemu zgodny z postawionymi celami.



Rysunek 4.1. Faza określania wymagań w cyklu życia oprogramowania

W przypadku firmy realizującej oprogramowanie na konkretne zamówienie, analitycy mają bezpośredni kontakt z przedstawicielami klienta. Podstawowym sposobem pracy jest więc wykonywanie wywiadów z różnymi osobami ze strony klienta. W odróżnieniu od fazy strategicznej, są to z reguły bezpośredni użytkownicy przyszłego systemu a nie zleceniodawcy. Faza ta wymaga dużego zaangażowania ze strony klienta. Najlepiej jest, jeżeli klient wyznacza osobę lub osoby, które będą odpowiedzialne za organizowanie współpracy analityków z przedstawicielami klienta. W wypadku dużych przedsięwzięć do tego celu wyznaczonych może zostać wiele osób, zawsze jednak klient powinien wyznaczyć konkretną osobę do pełnienia funkcji kierownika przedsięwzięcia ze strony klienta. Pomoc w realizacji fazy określania wymagań, jest dla przedstawicieli klienta dużym obciążeniem. Osoby te powinny być więc dodatkowo wynagradzane za wykonanie tej dodatkowej pracy. Wydatek taki jest niewielki w porównaniu z kosztem całego przedsięwzięcia i, biorąc pod uwagę duży koszt błędów popełnionych w tej fazie, jest z całą pewnością bardzo opłacalną inwestycją.

W przypadku firmy realizującej oprogramowanie sprzedawane rynkowo, korzystny jest również bezpośredni kontakt z osobami, które są przynajmniej potencjalnymi klientami i ekspertami z danej dziedziny. Faza określania wymagań wykonywana jest więc we współpracy z tymi osobami.

Trudność określania wymagań wynika z następujących czynników:

- Klient z reguły nie wie dokładanie w jaki sposób osiągnąć założone cele. Z drugiej strony cele klienta często można osiągnąć na wiele sposobów.
- Duże systemy są wykorzystywane przez wielu użytkowników. Ich cele są często sprzeczne. Różni użytkownicy mogą posługiwać się inną terminologią mówiąc o tych samych problemach.
- Zleceniodawcy i użytkownicy to często inne osoby. Głos zleceniodawców może być decydujący w tej fazie, choć nie zawsze potrafią oni właściwie przewidzieć potrzeby rzeczywistych użytkowników.

Wymagania klienta mogą być opisywane na różnych poziomach abstrakcji (ogólności):

- *Definicja wymagań* (ang. *requirement definition*), to ogólny opis w języku naturalnym. Opis taki jest wstępnym rezultatem rozmów z przedstawicielami klienta.
- *Specyfikacja wymagań* (ang. *requirements specification*), to częściowo ustrukturalizowany zapis wykorzystujący zarówno język naturalny, jaki i proste, częściowo przynajmniej sformalizowane notacje.
- *Specyfikacja oprogramowania* (ang. *software specification*), to w pełni formalny opis wymagań.

Specyfikacja oprogramowania jest rzadko wykonywana w praktyce. Często stosowanie formalnego opisu wymagań jest błędnie utożsamiane ze stosowaniem metody formalnych transformacji lub formalnymi dowodami poprawności. Formalna specyfikacja oprogramowania jest jednak także bardzo dobrą podstawą dla fazy testowania. Przykładem może być metoda Z schematów (ang. *Z schemas*) (Diller, 1990).

Dobry opis wymagań powinien:

- być kompletny oraz niesprzeczny,
- opisywać zewnętrzne zachowanie systemu a nie sposób jego realizacji,
- obejmować ograniczenia przy jakich musi pracować system,
- być łatwy w modyfikacji,
- brać pod uwagę przyszłe możliwe zmiany wymagań wobec systemu,
- opisywać zachowanie systemu w niepożądanych sytuacjach.

Najbardziej chyba podstawowym błędem popełnianym podczas określania wymagań jest koncentrowanie się na sytuacjach typowych i pomijanie wyjątków. Zarówno użytkownicy, jak i analitycy mają tendencję do nie zauważania sytuacji nietypowych.

Wymagania wobec oprogramowania można podzielić na dwa zasadnicze rodzaje:

- *Wymagania funkcjonalne*. Opisują one funkcje (czynności, operacje) wykonywane przez system.
- *Wymagania niefunkcjonalne*. Opisują ograniczenia, przy zachowaniu których system powinien realizować swoje funkcje.

Wymagania powinny zostać zebrane w dokumencie — opisie wymagań. Dokument ten powinien być podstawą formalnego, szczegółowego kontraktu między klientem a producentem. Powinien też pozwalać na weryfikację czy stworzony system rzeczywiście spełnia postawione wymagania. Powinien więc być to dokument zrozumiały dla obu stron. Należy jednak zaznaczyć, że producenci oprogramowania często nie są zainteresowani precyzyjnym opisem wymagań, który pozwoliłby na rzeczywistą weryfikację wyprodukowanego systemu.

Dokument opisujący wymagania powinien obejmować następujące części:

- *Wprowadzenie* — cele, zakres i kontekst systemu. Ta część dokumentu zawiera więc wyniki fazy strategicznej (być może częściowo zmodyfikowane).
- *Opis ewolucji systemu* — opis przewidywanych zmian wymagań wobec systemu.
- *Opis wymagań funkcjonalnych*.
- *Opis wymagań niefunkcjonalnych*.
- *Model systemu*, którego tworzenie opisane jest dokładniej w kolejnym rozdziale.
- *Słownik*. Opis wymagań może zawierać szereg terminów niezrozumiałych dla jednej ze stron. Mogą to być zarówno terminy informatyczne niezrozumiałe dla przedstawicieli klienta, jak i terminy dotyczące dziedziny zastosowania systemu niezrozumiałe dla przedstawicieli producenta (w szczególności dla osób niezaangażowanych bezpośrednio w fazy określania wymagań i analizy). Słownik powinien wyjaśniać tego typu terminy. Słownik powinien także precyzować terminy niejednoznaczne, które używane są w dokumencie w znaczeniu, będącym tylko jednym z wielu znaczeń, jakie mogą pojawić się innych kontekstach.

Dokument ten może zostać uzupełniony o następujące dodatki:

- Specyfikacja wymagań funkcjonalnych.
- Specyfikacja wymagań niefunkcjonalnych.
- Wymagania sprzętowe.
- Wymagania dotyczące bazy danych.
- Indeks pomocny w wyszukiwaniu w dokumencie konkretnych informacji.

Tworząc dokument opisujący wymagania należy przestrzegać następujących zasad:

- wymagania funkcjonalne powinny być oddzielone od wymagań niefunkcjonalnych,
- należy rozdzielać opisy poszczególnych wymagań, np. umieszczając opis każdego z nich w odrębnym akapicie,
- dla każdego wymagania powinien być podawany powód jego wprowadzenia — cel lub cele przedsięwzięcia, których osiągnięcie wspomaga dane wymaganie.

Określone w tej fazie wymagania będą podstawą testowania zrealizowanego systemu. W tej fazie powinien więc powstać także plan testów.

4.1. Wymagania funkcjonalne

Wymagania funkcjonalne opisują funkcje (czynności, operacje) wykonywane przez system. Funkcje te mogą być wykonywane przy udziale systemów zewnętrznych. Na przykład funkcja „Ewidencja podatników" realizowana przez system podatkowy, jest wykonywana przy udziale użytkownika, który wprowadza, usuwa i modyfikuje informacje o podatnikach.

Jak wspomniano powyżej język naturalny jest często stosowanym sposobem opisu wymagań. Stosowanie języka naturalnego ma jednak następujące wady:

- Niejednoznaczność języka naturalnego, która utrudnia precyzyjny zapis wymagań. Może ona także prowadzić do różnej interpretacji tego samego tekstu przez różne osoby.
- Elastyczność języka naturalnego, która pozwala te same treści wyrazić na różne sposoby. Utrudnia to wykrycie powiązanych wymagań. Może też prowadzić do przeoczenia sprzeczności wymagań sformułowanych w różny sposób lecz dotyczących tych samych funkcji.

Powyższych wad pozbawione są notacje formalne, które jednak są w zdecydowanej większości przypadków niezrozumiałe dla klienta. Mogą one więc być jedynie uzupełnieniem zapisu słownego.

Swego rodzaju kompromisem pomiędzy powyższymi metodami jest stosowanie formularzy do zapisu wymagań funkcjonalnych. Formularze takie wykorzystują język naturalny. Zapis jest jednak podzielony na konkretne pola, co pozwala uniknąć szeregu błędów w określaniu wymagań. Poniżej podany jest przykład formularza opisującego wymaganie funkcjonalne (tabela 4.1).

Definiując wymagania funkcjonalne z reguły podaje się jedynie wymagany efekt realizacji funkcji a nie jej algorytm. Jest więc to deklaratywny opis systemu. W szczególnych wypadkach wymagane może być jednak zastosowanie konkretnego algorytmu. Można uznać za wymaganie niefunkcjonalne związane z realizacją danej funkcji.

Tabela 4.1.
Przykład formularza opisu wymagania

Nazwa funkcji	*Edycja dochodów podatnika*
Opis	Funkcja pozwala edytować łączne dochody podatnika uzyskane w danym roku.
Dane wejściowe	Informacje o dochodach podatnika uzyskanych z różnych źródeł: kwoty przychodów, kosztów uzyskania przychodów oraz zapłaconych zaliczek na poczet podatku dochodowego. Informacje o dokumentach opisujących dochody z poszczególnych źródeł.
Źródło danych wejściowych	Dokumenty oraz informacje dostarczone przez podatnika. Dane wpisywane przez pracownika firmy podatkowej.
Wynik	
Warunek wstępny	Kwota dochodu = kwota przychodu – kwota kosztów (zarówno dla konkretnych dochodów, jaki i dla łącznych dochodów podatnika). Łączne kwoty przychodów, kosztów uzyskania, dochodów oraz zapłaconych zaliczek są sumami tych kwot dla dochodów z poszczególnych źródeł.
Warunek końcowy	Kwota dochodu = kwota przychodu – kwota kosztów (zarówno dla konkretnych dochodów, jaki i dla łącznych dochodów podatnika). Łączne kwoty przychodów, kosztów uzyskania, dochodów oraz zapłaconych zaliczek są sumami tych kwot dla dochodów z poszczególnych źródeł.
Efekty uboczne	Uaktualnienie podstawy opodatkowania.
Powód	Funkcja pomaga przyspieszyć obsługę klientów oraz zmniejszyć ryzyko popełnienia błędów.

Liczba wymagań funkcjonalnych może być bardzo duża. W literaturze można znaleźć informacje o systemach, w których wyróżniono nawet 8 000 różnych wymagań funkcjonalnych (Bell i in., 1977). Konieczne jest pewnego rodzaju uporządkowanie tych wymagań, które ułatwi pracę nad nimi. Poniżej omawiane są dwie, niewykluczające się zresztą metody, ułatwiające zapanowanie nad dużą liczbą wymagań. Są to: hierarchiczny zapis wymagań oraz diagramy przypadków użycia.

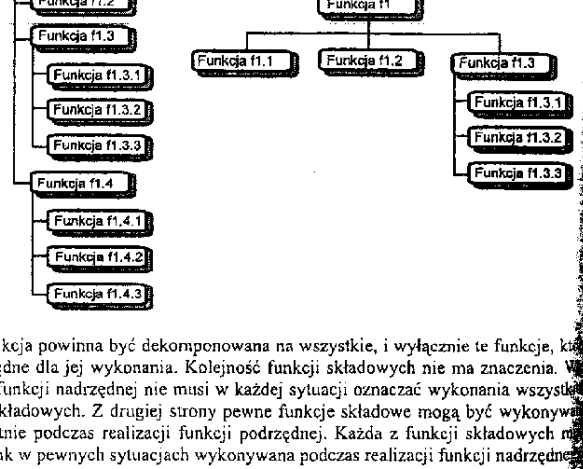
4.1.1. Hierarchia wymagań funkcjonalnych

Metoda ta opiera się na obserwacji faktu, że pewne funkcje wykonywane przez system są składowymi (podfunkcjami) innych bardziej ogólnych funkcji. Funkcja polegająca na projektowaniu systemu informacji geograficznej składa się z takich funkcji, jak: edycja tła, edycja obiektów umieszczonych na mapie, edycja słów kluczowych. Proponuje się więc, aby wymagania funkcjonalne zapisywać w formie hierarchicznej. Hierarchia taka może być opisana w formie tekstowej w sposób podany poniżej.

Tekstowy zapis hierarchii wymagań funkcjonalnych

Funkcja nadrzędna f1
- funkcja f1.1
- funkcja f1.2
- funkcja f1.3
  - funkcja f1.3.1
  - funkcja f1.3.2
  - funkcja f1.3.3
- funkcja f1.4
  - funkcja f1.4.1
  - funkcja f1.4.2
  - funkcja f1.4.3

Do zapisu hierarchii wymagań funkcjonalnych proponuje się także notacje graficzne, których przykłady prezentowane są na rysunku 4.2.



Rysunek 4.2. Przykłady graficznego zapisu hierarchii wymagań funkcjonalnych

Dana funkcja powinna być dekomponowana na wszystkie, i wyłącznie te funkcje, kt są niezbędne dla jej wykonania. Kolejność funkcji składowych nie ma znaczenia. W konanie funkcji nadrzędnej nie musi w każdej sytuacji oznaczać wykonania wszystk funkcji składowych. Z drugiej strony pewne funkcje składowe mogą być wykonywa wielokrotnie podczas realizacji funkcji podrzędnej. Każda z funkcji składowych m być jednak w pewnych sytuacjach wykonywana podczas realizacji funkcji nadrzędne

Wymagania funkcjonalne powinny dotyczyć wyłącznie zewnętrznych funkcji syste Z reguły oznacza to, że funkcje te powinny być widoczne dla użytkownika. Jeżeli

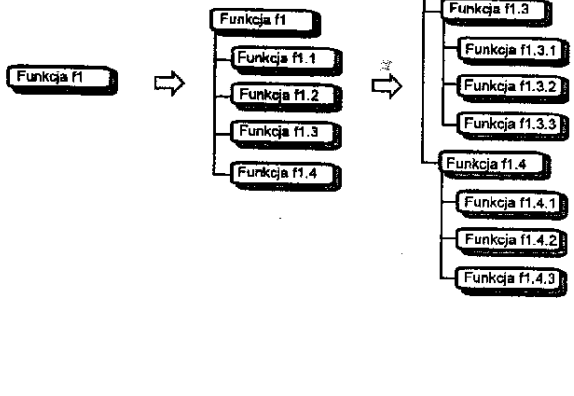
tem współpracuje z innymi systemami zewnętrznymi, może to także oznaczać widoczność tych funkcji np. dla innych programów. Funkcji systemu nie należy rozbijać do poziomu funkcji, które mają znaczenie czysto implementacyjne, nawet jeżeli poszczególne wymagania funkcjonalne są bardzo złożone i na etapie implementacji będą realizowane przez szereg składowych oprogramowania.

Pewne funkcje składowe mogą pojawiać się w ramach wielu funkcji nadrzędnych. Sytuacje takie powinny być wyróżniane w szczególny sposób. Przykład podany jest poniżej.

Kopie funkcji

Funkcja nadrzędna f1
- funkcja f1.1
  - funkcja f1.1.1
  - funkcja f1.1.2
- funkcja f1.2
  - funkcja f1.2.1
  - funkcja f1.2.2 (kopia funkcji f1.1.2)

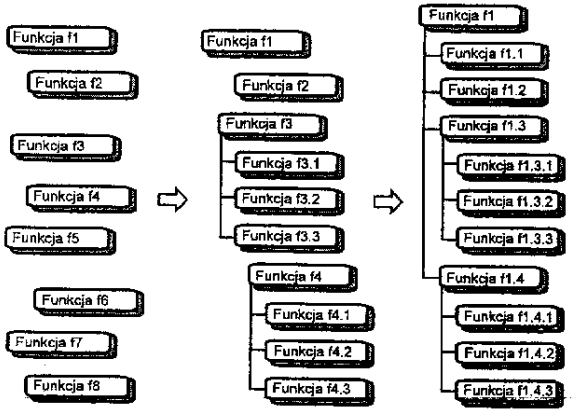
Jednym z proponowanych sposobów konstruowania hierarchii funkcji jest technika zstępująca (patrz rysunek 4.3). Zgodnie z tą techniką określanie wymagań funkcjonalnych należy rozpoczynać od poszukiwania najbardziej ogólnych funkcji. Następnie należy dekomponować te funkcje na funkcje składowe, aż do osiągnięcia poziomu funkcji elementarnych.



Rysunek 4.3. Zstępujące konstruowanie hierarchii wymagań

W praktyce nie zawsze możliwe jest kierowanie wywiadami z użytkownikami tak, aby rozpoczynać je od funkcji ogólnych a kończyć na najbardziej szczegółowych. Użytkownicy często nie są zainteresowani całością systemu i wolą koncentrować się na szczegółowych funkcjach, które uważają za najistotniejsze z ich punktu widzenia. Szczegółowe funkcje należy następnie agregować do funkcji wyższego poziomu. Taka technika konstruowania hierarchii funkcji nazywana jest techniką wstępującą (patrz rysunek 4.4).

Rysunek 4.4. Wstępujące konstruowanie hierarchii funkcji



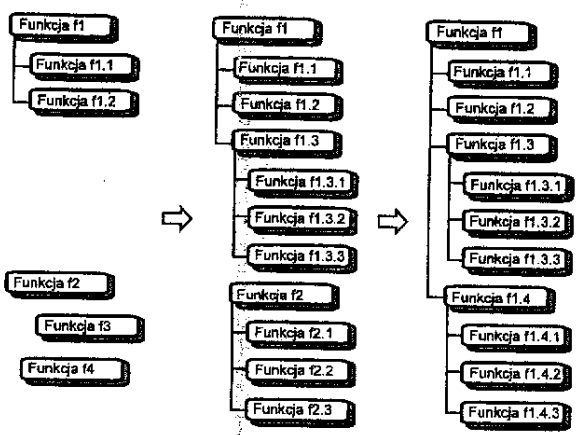
Najczęściej stosowane w praktyce podejście wykorzystuje obie z wymienionych powyżej technik. Analityk w miarę możliwości powinien starać się rozpoczynać określanie wymagań od funkcji ogólnych. Jeżeli jednak na danym etapie informacje uzyskane od użytkowników dotyczą szczegółowych funkcji, wprowadza się także bardziej szczegółowe wymagania. Hierarchia funkcji powstaje więc zarówno poprzez dekompozycję jak i agregację funkcji (patrz rysunek 4.5).

Poniżej podane są hierarchie funkcji dla rozważanych w książce przykładowych systemów. Rozwinięte są także opisy poszczególnych systemów. Dla celów książki opisy te zostały mocno uproszczone. Np. w systemie podatkowym rozważa się wyłącznie zeznania pojedynczego podatnika i zeznania małżeństwa, podczas gdy w rzeczywistości różnić można więcej typów zeznań.

System podatkowy

Program ułatwia przygotowywanie formularzy zeznań podatkowych (PIT-ów) oraz przechowywanie informacji o źródłach przychodów i ulg. Zeznanie może być tworzone dla pojedynczego podatnika lub dla małżeństwa. Zeznanie obejmuje informację o rocznych przychodach (w przypadku małżeństwa — z podziałem na przychody męża i żony) oraz o ulgach podatkowych. Przychody podzielone są na klasy ze względu na źródło uzyskania, np. pozarolnicza działalność gospodarcza, wolny zawód, ... W ramach danej klasy

Rysunek 4.5. Technika mieszana konstruowania hierarchii funkcji



przychodów podatnik mógł osiągnąć szereg przychodów z różnych źródeł (jeżeli np. pracuje lub wykonuje zlecenia dla wielu firm). Wszystkie przychody opisane są przez kwotę przychodu, kwotę kosztów, kwotę zapłaconych zaliczek oraz kwotę dochodu. Powyższe informacje pozwalają obliczyć należny podatek oraz kwotę do zapłaty/zwrotu. Zeznanie zawiera także informacje o podatniku(ach) oraz adres Urzędu Skarbowego do którego jest kierowane. System pozwala wydrukować wzorzec zeznania zawierający wszystkie informacje jakie podatnik musi umieścić w formularzu. Zeznanie można zabezpieczyć przed dalszymi zmianami (powinno to zostać zrobione po złożeniu zeznania w Urzędzie Skarbowym). System pozwala na tworzenie listy podatników oraz urzędów skarbowych, które mogą być pomocne podczas tworzenia nowego zeznania. Przechowuje także informacje o wszystkich przygotowanych zeznaniach.

Wspomaganie funkcjonowania firmy podatkowej (ogólna funkcja systemu)
- Ewidencja podatników
  - ...
- Ewidencja Urzędów Skarbowych
  - ...
- Ewidencja zeznań podatkowych
  - Dodawanie zeznania
  - Usuwanie zeznania
  - Zabezpieczanie zeznania
  - Edycja zeznania
    - Edycja informacji o przychodach
    - Edycja informacji o ulgach
    - ...
    - Wyświetlanie rozliczenia
    - Wydruk zeznania

Wyświetlanie rozliczenia (kopia)

Wydruk zeznania (kopia)

Przeglądanie zeznania (w wypadku zeznania zabezpieczonego możliwe jest jedynie przeglądanie jego opisu)

System informacji geograficznej — SIG

System informacji geograficznej jest rodzajem mapy komputerowej, składającej się z tła (na przykład zdjęcia satelitarnego, mapy fizycznej) oraz szeregu obiektów graficznych umieszczonych na tym tle. Obiekt może być punktem (np. budynek, firma), linią (rzeka, linia tramwajowa) lub obszarem (park, teren rekreacyjny). Każdy obiekt ma swoją nazwę oraz ewentualny opis. Opis ten może zostać wyświetlony na żądanie użytkownika. Każdy obiekt można opisać szeregiem słów kluczowych (np. jednostka budżetowa, firma komputerowa, obiekt znajdujący w dzielnicy Rataje). Użytkownik ma możliwość wyświetlenia tylko tych obiektów, które opisano wybranymi przez niego słowami kluczowymi.

Zarządzanie SIG (ogólna funkcja systemu)

- Przeglądanie SIG
  - Wyświetlanie mapy (różnych obszarów w różnych powiększeniach)
  - Wybór/pomijanie słów kluczowych
  - Wyświetlanie opisu obiektu graficznego
- Projektowanie SIG
  - Edycja tła
  - Edycja obiektów graficznych
    - Dodawanie obiektu
    - Edycja obiektu
    - Usuwanie obiektu
  - Edycja słów kluczowych
    - Dodawanie słowa kluczowego
    - Edycja słowa kluczowego
    - Usuwanie słowa kluczowego
  - Przeglądanie SIG (kopia)

System harmonogramowania zleceń

Zlecenia dla wydziału przygotowywane są przez dział marketingu. Zlecenie oznacza konieczność wyprodukowania konkretnej ilości pewnego wyrobu przed upływem pewnego konkretnego terminu. Czasami może być także określony najwcześniejszy pożądany termin realizacji. Ponieważ dział marketingu wykorzystuje własny system informatyczny, w którym między innymi umieszczane są informacje o zleceniach, pożądane jest, aby system harmonogramowania zleceń automatycznie odczytywał te informacje. Wyprodukowanie danego wyrobu (czyli realizacja pojedynczego zlecenia) wymaga wykonania ciągu operacji. Pewne operacje mogą być wykonywane wyłącznie na jednym urządzeniu, w wypadku innych możliwe jest wykorzystanie jednej z kilku maszyn, które mogą różnić się efektywnością pracy. Po wykonaniu pewnych operacji może być konieczna

przerwa, zanim rozpocznie się realizacja kolejnych operacji, z drugiej strony w pewnych wypadkach kolejne operacje muszą być wykonane przed upływem pewnego czasu (na przykład po gazowej sterylizacji pewnych wyrobów konieczna jest kilkugodzinna przerwa na ich odgazowanie, z drugiej strony, wyroby te muszą zostać szczelnie zamknięte przed upływem pewnego czasu po sterylizacji). Przestawienie danej maszyny na wykonywanie innej operacji może się wiązać z narzutem czasu. Co pewien czas (z reguły co miesiąc, chyba że pojawią się nowe pilne zlecenia) ustalany jest harmonogram niezrealizowanych zleceń. System powinien opracowywać harmonogramy w formie łatwej do wykorzystania przez kadrę kierowniczą wydziału oraz przygotowywać zamówienia dla magazynu na półprodukty. Zlecenia wykonane są usuwane ze zbioru niezrealizowanych zleceń.

Hierarchia funkcji:

- Zarządzanie zleceniami (ogólna funkcja systemu)
  - Ewidencja zleceń
    - Wczytywanie informacji o zleceniach
    - Wyświetlanie informacji o zleceniach
    - Wydruk informacji o zleceniach
  - Edycja technologicznego opisu wydziału
    - Edycja opisu maszyn
    - Edycja opisu operacji
    - Edycja opisu wyrobów
    - Sprawdzanie kompletności opisu
    - Wydruk informacji technologicznych
  - Harmonogramowanie zleceń
    - Ustalanie harmonogramu
    - Edycja harmonogramu
    - Graficzna prezentacja harmonogramu
    - Wydruk harmonogramu
  - Przygotowywanie kart zadań
  - Przygotowywanie zamówień na półprodukty

4.1.2. Diagramy przypadków użycia

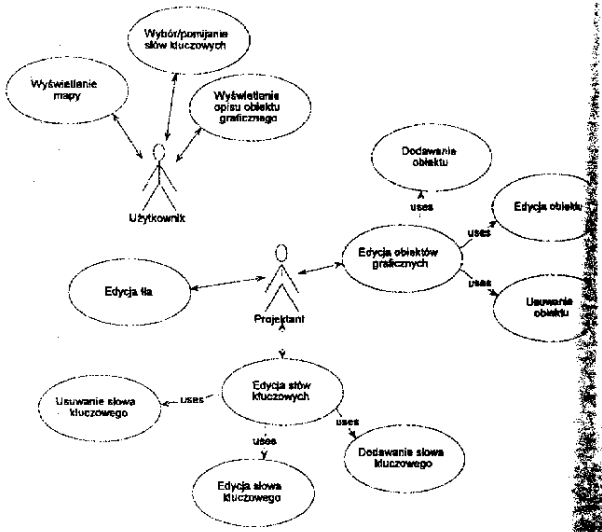
Omawiana w tym rozdziale metoda porządkowania wymagań funkcjonalnych opiera się na obserwacji, że systemy komputerowe wykorzystywane są przez różnych użytkowników korzystających z różnych funkcji systemu. Co więcej, nawet w przypadku bardzo złożonych systemów liczba funkcji wykorzystywanych przez poszczególnych użytkowników bywa stosunkowo nieduża. Obserwacja ta odnosi się zresztą również do innych rodzajów systemów zewnętrznych, np. innych programów. Opis wymagań poszczególnych użytkowników jest więc często znacznie prostszy niż opis wszystkich wymagań wobec systemu.

Wprowadzona przez Jacobsona (1993) metoda przypadków (sposobów) użycia (ang. *use cases*) rozpoczyna się od podziału systemów zewnętrznych współpracujących z systemem na klasy. W szczególności wyróżniane są klasy użytkowników wykorzystu-

jących system w podobny sposób. Należy zauważyć, że klasa użytkowników może obejmować wiele osób. Np. klasa „sekretarka" może obejmować wiele osób wykorzystujących proste funkcje systemu. Z drugiej strony klasy mogą odpowiadać rozmaitym rolom odgrywanym przez pewne osoby. Np. system informacji geograficznej będzie wykorzystywany przez „projektanta" oraz „osobę przeglądającą mapę". Konkretna osoba projektująca mapę będzie oczywiście wielokrotnie odgrywała rolę „osoby przeglądającej mapę". Klasa użytkowników może więc obejmować wiele konkretnych osób, konkretna osoba może natomiast odgrywać role odpowiadające wielu klasom.

Kolejnym krokiem jest identyfikacja funkcji istotnych dla poszczególnych klas systemów zewnętrznych. Tworzony jest więc opis systemu z punktu widzenia poszczególnych klas. Wyróżnienie klas użytkowników ułatwia organizowanie rozmów z przedstawicielami klienta. Przeprowadzając wywiad z konkretną osobą należy koncentrować się na funkcjach istotnych z punktu widzenia roli (ról) odgrywanych przez tę osobę. Jeżeli liczba funkcji wykorzystywanych przez daną klasę systemów zewnętrznych jest duża, można oczywiście skonstruować hierarchię tych wymagań.

Przykładowy diagram przypadków przedstawiony jest na rysunku 4.6.



Rysunek 4.6. Diagram przypadków użycia dla systemu informacji geograficznej

Stosowanie przypadków użycia nie jest oczywiście sprzeczne z hierarchicznym g wymagań. Funkcje systemu wysokiego poziomu grupują często funkcje wykorzy ne przez pewną klasę systemów zewnętrznych. Np. dwie główne funkcje systemu

macji geograficznej opisane w punkcie 4.1.1, tj. „przeglądanie SIG", i „projektowanie SIG", grupują funkcje wykorzystywane odpowiednio przez „osobę przeglądającą mapę" oraz „projektanta".

4.2. Wymagania niefunkcjonalne

Wymagania niefunkcjonalne opisują ograniczenia, przy których system musi realizować swoje funkcje. Można je podzielić na:

- Wymagania dotyczące produktu.

  Na przykład musi istnieć możliwość przeglądania systemu informacji geograficznej wyłącznie za pomocą klawiatury.

- Wymagania dotyczące procesu.

  Na przykład proces realizacji systemu harmonogramowania zleceń musi być zgodny ze standardem opisanym w dokumencie XXXA/96.

- Wymagania zewnętrzne.

  Na przykład system harmonogramowania zleceń musi współpracować z bazą danych systemu komputerowego działu marketingu opisaną w dokumencie YYYB/95. Niedopuszczalne są żadne zmiany w strukturze tej bazy.

Dobrze sformułowane wymagania niefunkcjonalne powinny być przede wszystkim weryfikowalne, tzn. musi istnieć możliwość sprawdzenia czy zrealizowany system rzeczywiście je spełnia. Wymagania stwierdzające, że system powinien być łatwy w obsłudze, niezawodny, wydajny nie mogą być obiektywnie zweryfikowane. Konieczne jest więc posługiwanie się wielkościami mierzalnymi dla wyrażania tego typu wymagań. Tabela 4.2 zawiera przykłady miar, które mogą służyć do ilościowego opisu pewnych wymaganych cech systemu.

Tabela 4.2
Tabelaryczny zapis rozważanych rozwiązań

Cecha	*Miary*
Wydajność	Liczba transakcji obsłużonych w ciągu sekundy Czas odpowiedzi Szybkość odświeżania ekranu
Rozmiar	Wymagana pamięć RAM Wymagana pamięć dyskowa
Łatwość użytkowania	Czas niezbędny dla przeszkolenia użytkowników Liczba stron dokumentacji

Tabela 4.2. (ciąg dalszy)
Tabelaryczny zapis rozważanych rozwiązań

Cecha	*Miary*
Niezawodność	Prawdopodobieństwo błędnego wykonania podczas realizacji transakcji
	Częstotliwość występowania błędnych wykonań
	Średni czas pomiędzy błędnymi wykonaniami
	Dostępność (procent czasu, w którym system jest dostępny)
Odporność (ang. *robustness*)	Czas restartu po awarii systemu
	Prawdopodobieństwo zniszczenia danych w przypadku awarii
Przenośność	Procent kodu zależnego od platformy docelowej
	Liczba platform docelowych
	Koszt przeniesienia na nową platformę

4.3. Podsumowanie

4.3.1. Kluczowe czynniki sukcesu

Najistotniejsze elementy wpływające na sukces fazy określania wymagań to:

- *Zaangażowanie właściwych osób ze strony klienta.*
- *Pełne rozpoznanie wymagań, wykrycie sytuacji szczególnych i nietypowych.* Typowy błąd popełniany w tej fazie to koncentrowanie się na sytuacjach typowych.
- *Sprawdzenie kompletności i spójności wymagań.* Przed przystąpieniem do dalszych prac, wymagania powinny zostać przejrzane pod kątem ich kompletności i wzajemnej spójności.
- *Określenie wymagań niefunkcjonalnych w sposób możliwy do weryfikacji.*

4.3.2. Podstawowe rezultaty fazy określania wymagań

Podstawowe rezultaty fazy określania wymagań to:

- dokument opisujący wymagania składający się z:
  - wprowadzenia opisującego cele, zakres i kontekst systemu,
  - opisu ewolucji systemu, to jest opisu przewidywanych zmian wymagań wobec systemu,
  - opisu wymagań funkcjonalnych,
  - opisu wymagań niefunkcjonalnych,
  - modelu systemu,
  - słownika,

  oraz następujących opcjonalnych dodatków:
  - specyfikacji wymagań funkcjonalnych,
  - specyfikacji wymagań niefunkcjonalnych,
  - opisu wymagań sprzętowych,
  - opisu wymagań dotyczących bazy danych,
  - indeksu,
- plan testów.

4.3.3. Narzędzia CASE w fazie określania wymagań

Wynikiem tej fazy jest spory zbiór danych, przede wszystkim informacji o poszczególnych wymaganiach. Narzędzia CASE ułatwiają przechowywanie, edycję oraz wyszukiwanie tych informacji. Dane te są przechowywane w specjalnej bazie danych, zwanej słownikiem danych (ang. *data dictionary*) lub repozytorium (ang. *repository*). Baza ta służy do przechowywania różnorodnych informacji dotyczących realizowanego przedsięwzięcia. Typowe informacje umieszczane w słowniku danych w fazie określania wymagań (częściowo także w fazie strategicznej) to:

- opisy wymagań funkcjonalnych i niefunkcjonalnych,
- opisy systemów zewnętrznych,
- plany testów.

Wiele narzędzi CASE pozwala użytkownikom w znacznym stopniu konfigurować strukturę słownika danych. Z reguły można modyfikować atrybuty za pomocą, których opisywane są poszczególne elementy. Jeżeli np. standardowa konfiguracja słownika danych nie obejmuje wszystkich atrybutów użytych do opisu wymagań funkcjonalnych w tabeli 4.1, z reguły istnieje możliwość dodania brakujących atrybutów. Rzadziej dopuszcza się dodawanie do słownika danych nowych typów elementów.

Narzędzia CASE obejmują także generatory raportów pozwalające na tworzenie rozmaitych raportów na podstawie zawartości słownika danych. Wiele narzędzi pozwala użytkownikowi na definiowane własnych raportów.

W fazie określania wymagań proponuje się również stosowanie notacji graficznych (porównaj rysunki 4.2 – 4.6). Narzędzia CASE zawierają więc specjalizowane edytory graficzne znacznie ułatwiające tworzenie diagramów tego typu. Zapewniają także zachowa-

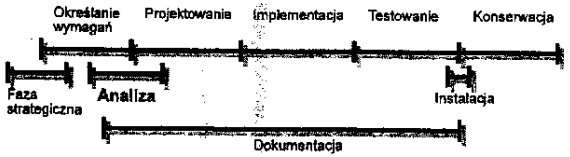
nie spójności pomiędzy słownikiem danych a diagramami graficznymi. Dodanie funkcji na diagramie graficznym powinno np. powodować automatyczne dodanie odpowiedniego wymagania do repozytorium. Z poziomu edytora graficznego można też odwoływać się do zawartości słownika danych. Podwójne kliknięcie na symbolu funkcji (przypadku użycia) może np. wyświetlać dialog służący do przeglądania i edycji opisu tej funkcji.

Dokumentacja będąca wynikiem tej fazy obejmuje zarówno raporty, jak i diagramy graficzne. Wiele narzędzi CASE zawiera moduły przygotowywania dokumentacji, umożliwiające przygotowywanie dokumentów obejmujących wybrane raporty, diagramy graficzne i dodatkowe informacje tekstowe.

Jak wspomniano w sekcji 2.3 budowa prototypu może być elementem fazy określania wymagań. Narzędzia CASE obejmują także składowe ułatwiające szybkie przygotowywanie prototypów: generatory kodu, narzędzia projektowania interfejsu użytkownika. Moduły te zostaną dokładniej omówione w dalszej części książki.

Rozdział 5.
Faza analizy (modelowania)

Zgodnie z rysunkiem 5.1 faza analizy pokrywa się częściowo zarówno z fazami określania wymagań, jak i projektowania. Ponieważ jednak metody, notacje i narzędzia służące modelowaniu są wyraźnie różne od stosowanych w tych dwóch fazach, analiza jest w niniejszej książce omawiana odrębnie.



Rysunek 5.1. Faza analizy w cyklu życia oprogramowania

Cel fazy analizy najlepiej określić poprzez porównanie z poprzedzającą ją fazą określania wymagań oraz następującą po niej fazą projektowania.

- Celem fazy określania wymagań jest udzielenie odpowiedzi na pytanie: co i przy jakich ograniczeniach system ma robić? Wynikiem tej fazy jest zbiór wymagań, czyli zewnętrzny opis systemu.
- Celem fazy projektowania jest udzielenie odpowiedzi na pytanie: jak system ma zostać zaimplementowany? Wynikiem jest projekt oprogramowania, czyli opis sposobu implementacji.
- Celem fazy analizy jest udzielenie odpowiedzi na pytanie: jak system ma działać? Wynikiem jest logiczny model systemu, opisujący sposób realizacji przez system postawionych wymagań, lecz abstrahujący od szczegółów implementacyjnych.

Ponieważ celem fazy analizy jest budowa logicznego modelu systemu jest ona także nazywana fazą modelowania.

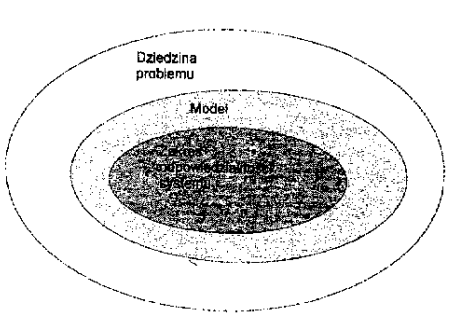
Logiczny model pozwala lepiej zrozumieć postawiony problem — i dzięki temu lepiej określić wymagania wobec systemu. Często więc w opisach cyklu życia oprogramowania wymienia się tylko jedną fazę, nazywaną fazą określania wymagań lub analizy obejmującą określanie wymagań i budowę modelu. Warto też dodać, że mianem analityków określa się zarówno osoby realizujące fazę określania wymagań, jak i fazę analizy.

Model systemu staje się podstawą tworzenia projektu. Zadania wykonywane w fazie analizy mogą być więc także utożsamiane z wyróżnianym tradycyjnie projektowaniem wysokiego poziomu, nie biorącym pod uwagę większości szczegółów implementacyjnych.

Tworzony system będzie stanowił fragment większej całości, zwanej *dziedziną problemu* (ang. *problem domain*). Przykłady dziedzin problemu, to: działalność firmy rachunkowej, funkcjonowanie wydziału firmy farmaceutycznej, wizualizacja informacji geograficznej. Fragment dziedziny problemu, który powinien zostać objęty przez tworzony system zwany jest *zakresem odpowiedzialności systemu* (ang. *system responsibilities*).

Tworzony w fazie analizy model z reguły wykracza poza zakres odpowiedzialności systemu (porównaj rysunek 5.2). Wynika to z następujących przyczyn:

- Ujęcie w modelu pewnych fragmentów dziedziny problemu nie będących częścią systemu czyni model bardziej zrozumiałym. Przykładem może być ujęcie w modelu systemów zewnętrznych, z którymi współpracuje system.
- Na etapie modelowania może nie być jasne, które elementy modelu będą realizowane przez stworzone oprogramowanie, a jakie na przykład w sposób sprzętowy lub ręcznie.
- W pewnych przypadkach może być jasne, że dostępne środki nie pozwolą na realizację całości systemu. Celem analizy może być między innymi wykrycie tych fragmentów dziedziny problemu, których wspomaganie za pomocą oprogramowania będzie szczególnie przydatne.



Rysunek 5.2. Dziedzina problemu, model i zakres odpowiedzialności systemu

Wiele systemów w prosty sposób automatyzuje wykonywanie pewnych czynności w danej organizacji nie zmieniając sposobu jej funkcjonowania. W tym wypadku tworzenie logicznego modelu systemu można utożsamiać z modelowaniem struktury oraz sposobu funkcjonowania tej organizacji. W innych sytuacjach wdrożenie oprogramowania może w istotny sposób zmienić dziedzinę problemu. Przykładem może być system harmonogramowania zleceń, pozwalający na zastosowanie złożonych algorytmów optymalizacji, które nie mogą być praktycznie stosowane bez wykorzystania komputerów. Celem analizy jest więc stworzenie modelu fragmentu dziedziny problemu zmodyfikowanej przez zastosowanie technik komputerowych.

Warto też wspomnieć, że omawiane w tym rozdziale metody i notacje wykorzystuje się także w ramach tzw. reinżynierii procesów biznesowych (ang. *business process reengineering*). Operacja ta rozpoczyna się od stworzenia modelu aktualnego sposobu funkcjonowania danej organizacji. Model ten poddawany jest następnie modyfikacjom, których celem jest poprawa sposobu realizowania przez tę organizację jej głównych funkcji. Modyfikacje te mogą, choć nie muszą, brać pod uwagę możliwości technik komputerowych. Dokładniejsze omówienie reinżynierii procesów biznesowych wykracza jednak poza zakres tej książki.

Składowe inżynierii oprogramowania wykorzystywane w fazie analizy to:

- notacje służące do zapisu modelu,
- metody budowy modelu (metody analizy),
- narzędzia ułatwiające stosowanie notacji i metod.

Należy podkreślić, że metody analizy nie są prostymi algorytmami, których stosowanie przez różne osoby zapewni osiągnięcie tych samych wyników. Proponowane metody są raczej zbiorem ogólnych wskazówek i rad. Analiza pozostaje w dużej mierze sztuką, której nie można nauczyć wyłącznie na podstawie literatury czy wykładów. Dla osiągnięcia prawdziwego zaawansowania w modelowaniu złożonych systemów niezbędne jest zdobyte w praktyce doświadczenie. Modele tego samego systemu stworzone przez różne osoby stosujące te same notacje i metody mogą się więc różnić. Z drugiej strony modele te nie powinny być zupełnie odmienne. Pewna nieprecyzyjność metod inżynierii oprogramowania, w szczególności metod analizy, bywa zresztą powodem krytyki ze strony osób uważających, że informatyka musi być dziedziną w pełni precyzyjną.

Rodzaje i role notacji wykorzystywanych w fazie analizy

Do zapisu modelu stosuje się następujące rodzaje notacji:

- język naturalny,

- notacje graficzne,
- specyfikacje — częściowo ustrukturalizowany zapis tekstowy i numeryczny.

Podobne notacje wykorzystywane są także w innych fazach realizacji przedsięwzięcia.

Szczególną rolę odgrywają notacje graficzne. Inżynieria oprogramowania wzoruje się tutaj na innych dziedzinach techniki (mechanice, elektronice), w których notacje graficzne są od dawna powszechnie stosowane. Zalety stosowania notacji graficznych potwierdzają także badania psychologiczne prowadzone między innymi w ramach prac nad sztuczną inteligencją.

Diagramy graficzne ułatwiają szybkie przekazywanie podstawowych informacji, nie pozwalają jednak na ukazanie wszystkich szczegółów. Nadmierne nagromadzenie szczegółów na diagramie graficznym może zresztą skutecznie zniweczyć jego czytelność. W związku z tym notacje graficzne uzupełniane są o dodatkowe informacje, tak zwane specyfikacje.

Najistotniejsze informacje powinny być oczywiście umieszczane na diagramach graficznych. Poszczególne metody różnią się między innymi tym, jakie informacje uznawane są za najważniejsze i prezentowane w sposób graficzny.

Wymienione powyżej notacje pełnią różnorodne funkcje:

- Są one podstawowym narzędziem pracy analityka i projektanta. Inżynier oprogramowania zapisuje i analizuje swoje pomysły, posługując się wymienionymi powyżej rodzajami notacji. Muszą więc one ułatwiać pracę nad złożonymi systemami. Powinny być więc przejrzyste, łatwo zrozumiałe, powinny ułatwiać modelowanie złożonych zależności.
- Notacje wykorzystywane w fazie analizy są często wykorzystywane do współpracy z użytkownikiem. Muszą więc być dość proste, przejrzyste, zrozumiałe.
- Praca inżyniera oprogramowania jest praktycznie zawsze pracą grupową. Notacje są podstawowym narzędziem współpracy pomiędzy członkami zespołu. Służą szybkiemu i precyzyjnemu przekazywaniu idei.
- Notacje wykorzystywane w fazie analizy i projektowania są podstawą implementacji oprogramowania. Powinny być więc przejrzyste i precyzyjne.
- Notacje te służą także do zapisu dokumentacji technicznej.

Na uwagę zasługuje rola notacji wykorzystywanych w fazie analizy i projektowania jako sposobu zapisu dokumentacji technicznej. Wielokrotnie spotykałem się z zainteresowaniem inżynierią oprogramowania wynikającym z poszukiwania notacji służących do zapisu dokumentacji technicznej. Wiele firm spostrzega konieczność zmiany sposobu pracy nie w trakcie budowy oprogramowania, lecz w trakcie jego eksploatacji, kiedy pojawia się konieczność dokonywania zmian w oprogramowaniu. Staje się wtedy jasne, że brak dokumentacji technicznej zdecydowanie utrudnia wykonywanie tych modyfikacji. Należy jednak podkreślić, że podobnie jak inne dziedziny techniki, inżynieria

oprogramowania raczej nie proponuje specjalnych notacji służących do opisywanie już istniejącego produktu. Dokumentacja techniczna budynku lub układu elektronicznego powstaje na bazie wcześniejszego projektu, być może po pewnych drobnych modyfikacjach. Podobnie techniczna dokumentacja oprogramowania powinna być wynikiem prac przeprowadzonych w fazach analizy i projektowania i zapisana za pomocą tych samych notacji.

5.2. Obiektowe i strukturalne metody analizy

Istnieją dwie główne grupy metod analizy. Tradycyjne metody, rozwijane od lat sześćdziesiątych, nazywane są metodami strukturalnymi (ang. *structured methods*). Metody te opierają się na wyróżnianiu w analizowanym systemie dwóch rodzajów składowych: składowych pasywnych odzwierciedlających fakt przechowywania w systemie pewnych danych oraz składowych aktywnych odzwierciedlających fakt wykonywania w systemie pewnych operacji. Metody obiektowe pojawiły się w latach osiemdziesiątych i są nadal intensywnie rozwijane. Ich podstawą jest wyróżnianie w systemie składowych, które łączą w sobie możliwość przechowywania danych oraz wykonywania pewnych operacji.

Analiza strukturalna (ang. *structured analysis*) rozpoczyna się od budowy dwóch różnych modeli systemu: modelu danych będącego opisem pasywnej części systemu oraz modelu funkcji opisującego aktywną część systemu. Te dwa modele są następnie integrowane. Wynikiem tej integracji jest model przepływów danych (ang. *data flows model*). Zwolennicy metod strukturalnych argumentują, że budowa dwóch różnych modeli systemu ułatwia zrozumienie jego różnorodnych aspektów. Krytycy tych metod zwracają uwagę na to, że integracja tych dwóch różnych modeli może być bardzo trudna, w szczególności wtedy, gdy modele te są wykonywane przez różne zespoły. Metody strukturalne są szczególnie przydatne w przypadkach, gdy jeden z aspektów pasywny lub aktywny jest wyraźnie bardziej istotny niż drugi. W przypadku systemów informatycznych służących niemal wyłącznie do przechowywania, i wyszukiwania złożonych danych, nie realizujących żadnych złożonych funkcji, najistotniejsza jest budowa modelu danych. Innym skrajnym przypadkiem są systemy realizujące złożone funkcje na prostych danych (na przykład złożone obliczenia naukowe). W tym wypadku istotniejsze jest modelowanie funkcji. Metody strukturalne sprawdzają się gorzej, jeżeli system ma realizować skomplikowane funkcje na złożonych danych.

Przymiotnik obiektowy stał się w ostatnich latach bardzo popularny i rozumiany na różne sposoby. Warto więc określić, jak będzie on rozumiany w tej książce. Dany system jest analizowany w sposób obiektowy jeżeli:

- jest dzielony na obiekty, posiadające:
  - tożsamość (ang. *identity*), czyli dające się jednoznacznie wyróżnić w ramach systemu,

- stan (ang. *state*),
- zachowanie (ang. *behavior*),

☞ obiekty są grupowane w klasy złożone z obiektów o podobnych stanach i zachowaniu.

Obiektowe metody analizy (ang. *object-oriented methods*) zyskują w ostatnich latach dużą popularność. Wynika to z dwóch głównych przyczyn. Metody te są wygodnym narzędziem analizy złożonych systemów, w szczególności systemów o złożonej stronie pasywnej oraz złożonej funkcjonalności. Z drugiej strony popularność tych metod wiąże się z rosnącą popularnością obiektowych środowisk implementacji. Analiza i projektowanie strukturalne nie są dobrą podstawą dla implementacji w środowisku obiektowym. Programowanie obiektowe ułatwia:

☞ ukrywanie danych (hermetyzację) (ang. *encapsulation*),

☞ wykorzystanie gotowych elementów,

☞ ponowne wykorzystanie oprogramowania (ang. *software reuse*),

☞ szybkie prototypowanie (ang. *rapid prototyping*),

☞ programowanie oparte na zdarzeniach (ang. *event-driven programming*),

☞ programowanie przyrostowe (ang. *incremental programming*).

Stosowanie obiektowych metod analizy nie musi wiązać się ze stosowaniem w dalszych fazach obiektowych środowisk implementacji (obiektowych języków programowania, obiektowych baz danych, bibliotek obiektowych). Model i powstały na jego bazie projekt obiektowy można zaimplementować stosując np. języki proceduralne i relacyjne bazy danych. Implementacja ta nie jest przy tym istotnie trudniejsza niż implementacja projektu strukturalnego. Z drugiej strony większość typowo stosowanych obiektowych języków programowania to języki hybrydowe pozwalające na posługiwanie się konstrukcjami proceduralnymi. Stosując takie języki można bez trudu zaimplementować projekt strukturalny, wykorzystując nawet pewne elementy programowania obiektowego, np. dla definiowania nowych typów danych. Wiele (jeżeli nie większość) programów pisanych np. w C++ nie posiada w pełni wszystkich wymienionych powyżej cech systemu obiektowego (co nie oznacza automatycznie, że są to złe programy).

5.3. Notacje obiektowe i ich interpretacja

W literaturze można znaleźć szereg propozycji metod i notacji obiektowych. Najpopularniejsze z nich (tj. najczęściej wykorzystywane w narzędziach CASE) to:

☞ OMT (Object Modelling Technique), której głównym autorem jest Rumbaugh (Rumbaugh i in., 1991),

☞ OOA i OOD (Object-Oriented Analysis, Object-Oriented Design) Coada i Yourdona (Coad i Yourdon, 1994),

☞ OOAD (Object-Oriented Analysis and Design) Boocha (Booch, 1994).

Podejścia proponowane przez tych autorów różnią się częściowo, nie są jednak ze sobą sprzeczne, lecz raczej uzupełniają się nawzajem. Z drugiej strony poszczególne metody zawierają pewne elementy rzadko wykorzystywane w praktyce (np. model przepływów danych w OMT).

W niniejszej książce analiza obiektowa omawiana jest jako spójna metoda obejmująca najbardziej przydatne elementy wszystkich powyższych propozycji. Podobne podejście zastosowano także przy omawianiu analizy strukturalnej. Jest to umotywowane następującymi czynnikami:

☞ Podejścia proponowane przez różnych autorów różnią się częściowo, nie są jednak ze sobą sprzeczne, lecz raczej uzupełniają się nawzajem.

☞ Poszczególne metody zawierają często elementy rzadko wykorzystywane w praktyce (na przykład model przepływów danych w OMT).

☞ Notacje proponowane przez różnych autorów nie są nierozerwalnie związane z proponowaną metodyką. Posługując się np. notacją OMT można konstruować model zgodnie z metodyką Coada i Yourdona.

☞ W praktyce analitycy nie posługują się wyłącznie jedną metodyką, lecz wybierają techniki, które uznają w danym momencie za najbardziej przydatne.

☞ Narzędzia CASE nie narzucają stosowania konkretnej metodyki nawet jeżeli stosują wyłącznie jedną notację. Omawiane w dalszej części rozdziału notacje są najczęściej dostępne w obiektowych narzędziach CASE.

Podstawową notacją obiektową wykorzystywaną w książce jest notacja OMT.

5.3.1. Diagramy klas i obiektów

Diagramy klas i obiektów służą do prezentowania składowych systemu oraz zależności pomiędzy nimi. Przed przedstawieniem symboli wykorzystywanych na tych diagramach należy jednak zdefiniować pojęcia obiektu i klasy.

Obiekt (ang. *object*)

- Na etapie analizy obiekt oznacza składową dziedziny problemu posiadającą tożsamość, stan i zachowanie.
- Na etapie projektowania i implementacji pojęcie to oznacza konstrukcję języka programowania łączącą dane i metody (procedury, funkcje).

Klasa (ang. *class*)

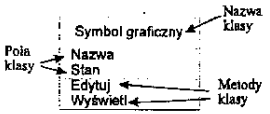
- Na etapie analizy klasa oznacza wzorzec grupy obiektów.
- Na etapie projektowania i implementacji pojęcie to oznacza typ obiektowy w języku programowania.

Jak widać pojęcia klasy i obiektu wykorzystywane są w fazach analizy, projektowania i implementacji. Jedną z zalet metod obiektowych jest właśnie fakt posługiwania się tymi samymi pojęciami we wszystkich tych fazach realizacji przedsięwzięcia, choć interpretacja tych pojęć jest różna w różnych fazach.

Klasa składa się z pól (atrybutów, ang. *fields*, *attributes*) i metod (usług, operacji, ang. *methods*, *services*, *operations*). Pola opisują stan obiektów klasy. Metody opisują operacje jakie można na polach tej klasy wykonywać.

Symbol klasy (patrz rysunek 5.3) składa się z trzech części. Części zawierające pola i metody mogą zostać pominięte.

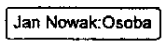
Rysunek 5.3.
Symbol klasy



Rysunek 5.4 przedstawia symbol obiektu. Obiekty umieszczane są najczęściej na odrębnych diagramach. Opis obiektu składa się z dwóch części nazwy obiektu i nazwy klasy. Jedna z tych części może zostać pominięta:

- Jan Nowak:Osoba — obiekt Jan Nowak klasy Osoba,
- :Osoba — nieokreślony obiekt klasy Osoba,
- Jan Nowak: — obiekt Jan Nowak nieokreślonej klasy.

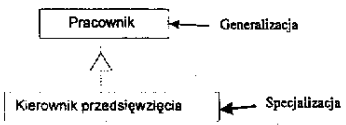
Rysunek 5.4.
Symbol obiektu



Związek generalizacji-specjalizacji (ang. *generalization-specialization link/association*)

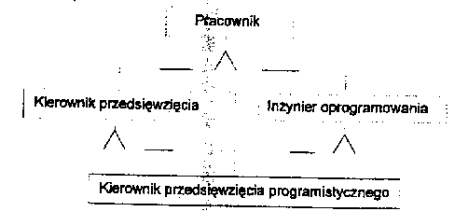
Dwie klasy pozostają w związku generalizacji-specjalizacji, jeżeli jedna z nich, zwana specjalizacją, jest rodzajem drugiej, nazywanej generalizacją. Generalizacja jest uogólnieniem specjalizacji.

Rysunek 5.5.
Symbol generalizacji-specjalizacji



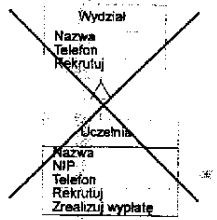
Generalizacja może mieć wiele specjalizacji. Specjalizacja może mieć wiele generalizacji (porównaj rysunek 5.6). Pierwszą z powyższych sytuacji można uznać za typową. Druga pojawia się znacznie rzadziej.

Rysunek 5.6.
Przykład wielu generalizacji i specjalizacji klasy



Związku generalizacji-specjalizacji, który opisuje pewien związek pomiędzy klasami w dziedzinie problemu, nie należy utożsamiać z dziedziczeniem, które jest konstrukcją w obiektowych językach programowania. Języki obiektowe pozwalają na deklarowanie nowej klasy na bazie innej (innych) klasy. Nowa klasa dziedziczy wtedy pola i metody klasy (klas) bazowych (niektóre języki obiektowe pozwalają ograniczyć dziedziczenie pewnych pól i metod). Stwierdzenie, że pewna klasa posiada wszystkie pola i metody innej klasy nie jest wystarczające do wprowadzenia związku generalizacji-specjalizacji. Przykład przedstawiony jest na rysunku 5.7. Na pewnym etapie analizy systemu wyróżniono klasy Wydział i Uczelnia, przy czym druga z tych klas zawiera wszystkie pola i metody pierwszej. Uczelnia nie jest jednak rzecz jasna rodzajem Wydziału. Jest bardzo prawdopodobne, że w pewnym momencie, być może już po opracowaniu pierwszej wersji systemu, pojawi się konieczność wprowadzenia w klasie Wydział pól i/lub metod nie mających sensu w klasie Uczelnia.

Rysunek 5.7.
Przykład błędnego [illegible] generalizacji-[illegible]

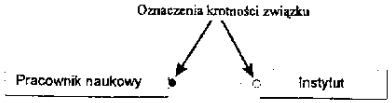


Związek klas (ang. *class link/assciations*)

Pomiędzy dwoma (trzema) klasami istnieje związek, jeżeli obiekty jednej z tych klas są w ramach dziedziny problemu w pewien sposób powiązane z obiektami pozostałych klas. Rysunek 5.8 przedstawia związek opisujący fakt, że pracownicy naukowi uczelni pracują w instytutach, z których składa się ta uczelnia. Dodatkowe symbole na końcach linii odpowiadającej związkowi opisują tzw. krotność związku (ang. association/link multiplicity), czyli liczby obiektów danej klasy, które mogą być powiązane z obiektami drugiej klasy. W przypadku przedstawionym na rysunku 5.8 dany pracownik naukowy

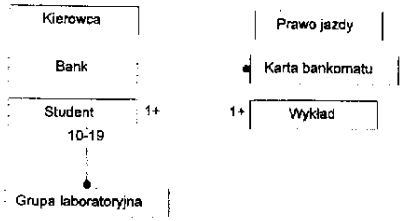
może lecz nie musi, pracować w jednym instytucie. Dany instytut może, lecz nie musi zatrudniać wielu pracowników naukowych (instytut nie zatrudnia pracowników bezpo średnio po utworzeniu).

Rysunek 5.8. *Symbol związku dwóch klas*



Przykłady związków o innych krotnościach podane są na rysunku 5.9.

Rysunek 5.9. *Przykłady związków klas*



Wyrażają one następujące zależności:

- Kierowca ma dokładnie jedno prawo jazdy. Prawo jazdy jest przyznane dokładnie jednego kierowcy.
- Bank wydaje zero lub wiele kart bankomatu. Karta bankomatu jest wydana przez dokładnie jeden bank.
- Student uczęszcza na wiele wykładów. Wykład jest prowadzony dla wielu studentów.
- Student może, lecz nie musi, być przypisany do wielu grup laboratoryjnych. Do grupy laboratoryjnej musi być przypisanych nie mniej niż dziesięciu i nie więcej niż dziewiętnastu studentów.

Oznaczenia krotności związków są zebrane w tabeli 5.1.

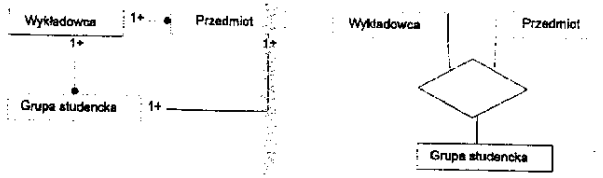
W wielu wypadkach w związku biorą udział obiekty trzech różnych klas. Przykład kiego związku jest pokazany na rysunku 5.10. Związek ten oznacza, że wykładow wykłada pewien przedmiot dla pewnej grupy studenckiej. Rysunek 5.10 pokazuje ta że związek taki można zamodelować w postaci trzech związków pomiędzy dwoma sami. Traci się jednak wtedy istotną informację o charakterze tego związku.

W pewnych sytuacjach może być oczywiste jaki związek zachodzi pomiędzy dw klasami. W innych sytuacjach, w szczególności wtedy, gdy pomiędzy tymi klasami chodzi więcej związków, konieczne jest opisanie związku.

Tabela 5.1. *Oznaczenia krotności związków klas*

Krotność	*Oznaczenie*
Dokładnie jeden	—
Zero lub wiele	—●
Jeden lub wiele	1+
Zero lub jeden	—○
Podana liczba	3
Zakres	3 – 7
Zakres lub liczba	3 – 7, 10

Rysunek 5.10. *Przykład związku ternarnego*



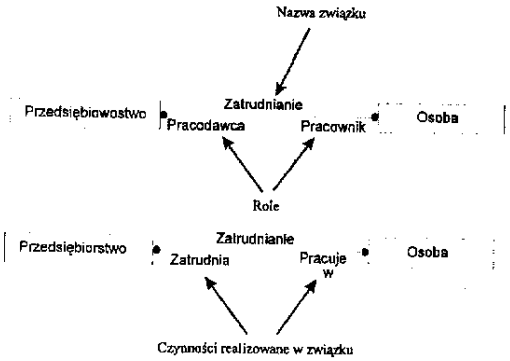
Związek klas można opisywać podając:

- nazwę związku, np. zatrudnianie, wykładanie,
- rolę (role) odgrywane w związku przez obiekty danej klasy,
- czasownik opisujący czynność (operację, zadanie) wykonywaną w ramach związku przez obiekt danej klasy.

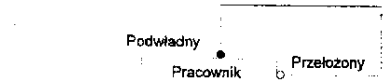
Rola odgrywana w związku przez obiekty danej klasy może być taka sama jak nazwa klasy. Przykłady opisu związków podane są na rysunku 5.11. W praktyce nie opisuje się związku korzystając ze wszystkich powyższych opcji. Podanie wyłącznie nazwy związku, jednej roli lub czynności z reguły jednoznacznie identyfikuje związek.

Związki mogą również zachodzić pomiędzy obiektami tej samej klasy (porównaj rysunek 5.12).

Na rysunku 5.9 podano między innymi przykład związku jaki zachodzi pomiędzy bankiem a kartą bankomatu. Wynika z niego, że bank wydaje zero lub wiele kart bankomatu. W rzeczywistości bank wydaje jednak tylko jedna kartę posiadającą pewien numer PIN (Personal Identification Number). W ramach danego banku numer PIN jednoznacznie identyfikuje więc konkretną kartę. Informację tę można zapisać za pomocą tak zwanych związków kwalifikowanych (ang. qualified association) (patrz rysunek 5.13). Numer PIN jest w tym wypadku kwalifikatorem związku (ang. association qualifier). Wprowadzenie kwalifikatora pozwoliło w tym wypadku ograniczyć krotność związku.



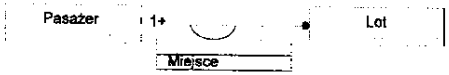
Rysunek 5.11. Przykłady opisów związków klas



Rysunek 5.12. Przykład związku pomiędzy obiektami tej samej klasy

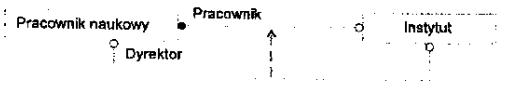
Rysunek 5.13. Związek kwalifikowany

W przypadku pewnych związków pojawia się konieczność opisania ich za pomocą p które nie są jednak polami żadnej z klas. Przykładowo pasażer lecący pewnym lot zajmuje konkretne miejsce. Miejsce nie opisuje jednak ani lotu ani pasażera, jest na miast opisem samego związku. Sposób zapisu pól związków podano na rysunku 5.14.



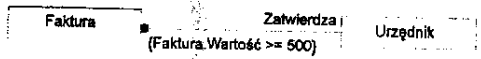
Rysunek 5.14. Pole związku

Obiekty mogą pozostawać w pewnych związkach tylko jeżeli zachodzi równocze pewien inny związek. Dyrektor instytutu naukowego musi być jednocześnie jego cownikiem. Sposób zapisu takiego ograniczenia jest pokazany na rysunku 5.15.



Rysunek 5.15. Ograniczenie pomiędzy związkami

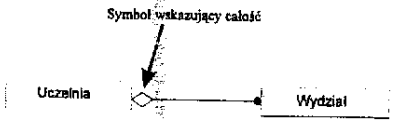
Możliwość zajścia pewnego związku może być także uzależniona od innych warun na przykład od wartości pól powiązanych obiektów. Na rysunku 5.16 podany jest kład związku pomiędzy urzędnikiem a fakturą, która musi być zatwierdzona, jeże wartość przekracza 500 zł.



Rysunek 5.16. Ograniczenie dotyczące związku

Związek agregacji (ang. *aggregation*)

Związek ten oznacza, że obiekty pewnej klasy składają się z obiektów innych klas lub tej samej klasy. Symbol związku agregacji pokazano na rysunku 5.17. Symbol ten także zawiera informacje o krotności związków. Związek ten nazywany jest także związkiem całość-część (ang. *whole-part relationship*).



Rysunek 5.17. Symbol związku agregacji

5.3.2. Diagramy interakcji

Przed omówieniem diagramów interakcji (ang. *interaction diagrams*) konieczne jest wprowadzenie pojęcia komunikatu.

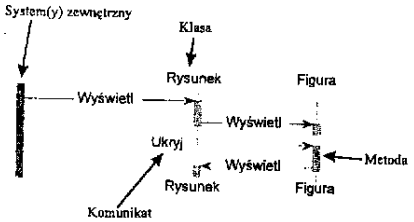
Komunikat (ang. *message*)

Komunikat wysyłany do obiektu pewnej klasy oznacza żądanie wykonania jednej z metod tej klasy, jest więc wywołaniem pewnej metody. Komunikat może być wysyłany przez system zewnętrzny lub przez obiekt jednej z klas systemu. W tym drugim wypadku komunikat jest wysyłany w trakcie wykonywania jednej z metod klasy, która jest nadawcą komunikatu. Należy zaznaczyć, że wysłanie komunikatu nie kończy realizacji metody, w ramach której został on wysłany.

Wysłanie komunikatu może wiązać się z przekazaniem pewnych danych wejściowych do wywoływanej metody oraz z pobraniem danych wyjściowych tej metody. Nazwą komunikatu jest nazwa wywoływanej metody.

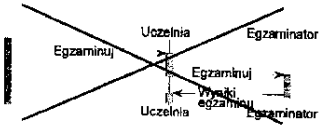
Diagram interakcji przedstawia pewien scenariusz przepływu komunikatów pomiędzy obiektami systemu oraz systemami zewnętrznymi. Opisują one sposób w jaki obiekty współpracują ze sobą w celu zrealizowania funkcji systemu. Diagram interakcji tworzy się więc dla pewnej funkcji systemu (niekoniecznie elementarnej). Diagram interakcji może też opisywać sposób realizacji pewnej złożonej metody, która wymaga wywołania metod z innych obiektów. Przykład diagramu interakcji podany jest na rysunku 5.18.

Diagram interakcji prezentuje potencjalny scenariusz przepływu komunikatów. W konkretnej sytuacji pewne komunikaty mogą nie być wysyłane. Przykładowo w sytuacji przedstawionej na rysunku 5.18, klasa Rysunek nie wyśle komunikatu Wyświetl do klasy Figura, jeżeli rysunek jest pusty. Diagram interakcji nie zawiera też informacji o tym czy dany komunikat jest wysyłany tylko raz czy wielokrotnie.



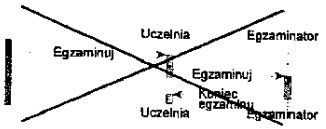
Rysunek 5.18.
Przykład diagramu interakcji

Warto w tej chwili zwrócić uwagę na dwa podstawowe błędy popełniane przy tworzeniu diagramów interakcji. Pierwszy z tych błędów polega na traktowaniu komunikatu jako przepływu danych. Przykład pokazany jest na rysunku 5.19.



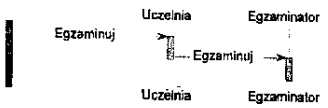
Rysunek 5.19.
Symbol komunikatu błędnie użyty do zobrazowania przepływu danych

Egzaminator po przeprowadzeniu egzaminu zwraca wyniki egzaminu. Są to jednak dane wyjściowe metody Egzaminuj. Przykład kolejnego błędu, który może pojawić się w tej samej sytuacji, pokazany jest na rysunku 5.20. Komunikat Koniec egzaminu został błędnie wprowadzony po to, aby zobrazować fakt, że po zakończeniu wykonywania metody Egzaminuj klasy Egzaminator dalsze operacje wykonywane są przez klasę Uczelnia. Symbol komunikatu użyto więc dla przedstawienia zdarzenia polegającego na zakończeniu realizacji metody Egzaminuj. Nie jest to potrzebne ponieważ, jak wspomniano powyżej, wywołanie metody Egzaminuj klasy Egzaminator nie kończy wykonywania metody Egzaminuj klasy Uczelnia.



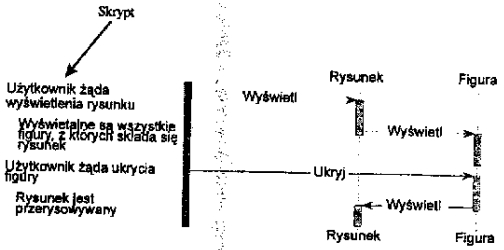
Rysunek 5.20.
Symbol komunikatu błędnie użyty do zobrazowania zdarzenia

Należy wspomnieć, że ramach pewnych metod obiektowych (OMT i OOAD) proponuje się stosowanie analogicznych diagramów, na których prezentuje się przepływy zdarzeń. Podejście to nie jest omawiane w ramach tej książki. Poprawny sposób zobrazowania powyższej sytuacji pokazany jest na rysunku 5.21.



Rysunek 5.21.
Poprawny diagram interakcji

Informacje zawarte na diagramie interakcji mogą być uzupełnione tzw. skryptem. Jest to tekstowy opis czynności realizowanych w ramach scenariusza zawartego na diagramie. Skrypt pozwala między innymi zawrzeć informacje o opcjonalności pewnych komunikatów oraz o liczbie wysyłanych komunikatów. Skrypt pozwala również dokładniej opisać czynności wykonywane przez poszczególne metody. Przykład diagramu opisanego skryptem przedstawiony jest na rysunku 5.22.



Rysunek 5.22.
Diagram interakcji opisany skryptem

5.3.3. Diagramy przejść stanów

Diagramy przejść (transformacji) stanów (ang. state transition diagrams) służą do prezentowania dynamicznych, tj. zmiennych w czasie aspektów systemu. Służą one do prezentowania dynamicznego zachowania całego systemu, grup klas lub pojedynczych klas o złożonej dynamice. Ze względu na swą elastyczność mogą też, jako alternatywa dla diagramów interakcji, służyć do prezentowania sposobu realizacji funkcji systemu. Na wstępie definiowane są podstawowe pojęcia występujące na tych diagramach.

Zdarzenie (ang. *event*)

Zdarzenie to zjawisko, które zachodzi w pewnym punkcie czasu (w ramach sensownego przybliżenia) i ma wpływ na modelowany system. Przykłady zdarzeń to:

- odjazd pociągu do Gdańska,
- wprowadzenie danych,
- wybranie polecenia z menu,
- przekroczenie temperatury 50°C.

Zdarzenie zachodzące poza systemem nazywane jest *zdarzeniem zewnętrznym*. Przykłady zdarzeń zewnętrznych to:

- wprowadzenie danych,
- wybranie polecenia z menu,
- przerwanie przez użytkownika wykonywania operacji.

Zdarzenie wewnętrzne ma swoje źródło w ramach systemu. Przykłady takich zdarzeń to:

- zakończenie wykonywania metody,
- błąd arytmetyczny,
- przekroczenie czasu.

Pod pojęciem zdarzenia rozumie się pewną klasę zjawisk, na które system reaguje w podobny sposób. Poszczególne wystąpienia zdarzenia *żądanie otworzenia dokumentu* mogą się na przykład różnić nazwą pliku, który należy odczytać. Jest to przykład zdarzenia, które jest opisane dodatkowymi parametrami.

Stan (ang. *state*)

Stan jest okresem czasu ograniczonym przez dwa zdarzenia. System (fragment systemu) znajdując się w różnych stanach reaguje w sposób jakościowo różny na zachodzące zdarzenia. W ramach danego stanu reakcje są jakościowo takie same i nie zależą od sposobu w jaki stan ten został osiągnięty. Edytor tekstu znajdujący się w stanie edycji dokumentu reaguje na polecenie drukowania przejściem do stanu drukowania dokumentu. Zachowanie to uznaje się za jakościowo takie same niezależnie od zawartości edytowanego dokumentu (mimo, że efekt na papierze jest za każdym razem inny).

Przejście (transformacja) (ang. *transition*)

Zdarzenie może spowodować zmianę stanu systemu (fragmentu systemu). Zakłada się, że przejście zachodzi natychmiastowo (w ramach sensownego przybliżenia. Przejście może być uwarunkowane spełnieniem pewnego dodatkowego warunku.

Akcja (ang. *action*)

Jest to czynność wykonywana w momencie zajścia zdarzenia. Zakłada się, że jest ona wykonywana natychmiast (w ramach sensownego przybliżenia). Przykładowe akcje to:

- ustawienie wartości pól,
- wykonanie obliczeń.

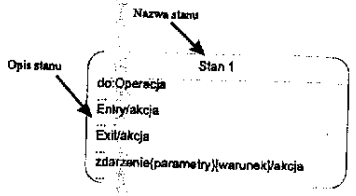
Operacja (ang. *activity*)

Jest to czynność, której wykonanie trwa przez pewien niepomijalny okres. O ile akcja była wykonywana w momencie zajścia zdarzenia, to operacja jest wykonywana w sie kiedy system (fragment systemu) znajduje się w pewnym stanie. Operacja może stać przerwana w momencie zajścia zdarzenia, które powoduje wyjście z tego stanu. Jeżeli operacja kończy się samoczynnie, generuje w momencie swojego zakończenia rzenie, które powoduje przejście do innego stanu. Przykładowe operacje to:

- monitorowanie czujnika,
- wyświetlanie aktualnego czasu,
- wykonanie złożonych obliczeń.

Symbol stanu przedstawiony jest na rysunku 5.23.

Rysunek 5.23. *Symbol stanu*



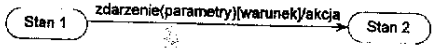
Po słowie kluczowym do wymienia się operację wykonywaną w tym stanie. W danym stanie może być wykonywanych wiele operacji. Po słowie kluczowym Entry wymienia się akcję wykonywaną w momencie wejścia do stanu, niezależnie od tego jakie zdarzenie spowodowało przejście. Po słowie kluczowym Exit wymienia się akcję wykonywaną w momencie wyjścia ze stanu, niezależnie od tego, jakie zdarzenie spowodowało przejście. Zarówno w momencie wejścia, jak i wyjścia ze stanu może być wykonywanych wiele akcji. W ramach opisu stanu, wymienia się też zdarzenia, których zajście, w czasie gdy system przebywa w tym stanie, powoduje wykonanie pewnej akcji nie powodując przejścia do innego stanu.

Wyróżnione symbole służą do przedstawiania początkowego i końcowego stanu systemu (patrz rysunek 5.24).

5.24.
stanu
ego

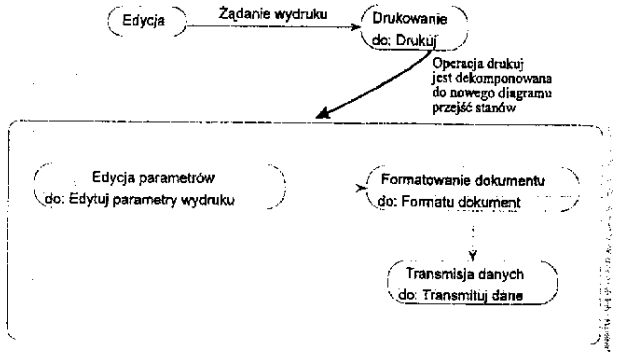
Symbol przedstawiający przejście ze stanu do stanu umieszczony jest na rysunku 5.25.

zenia



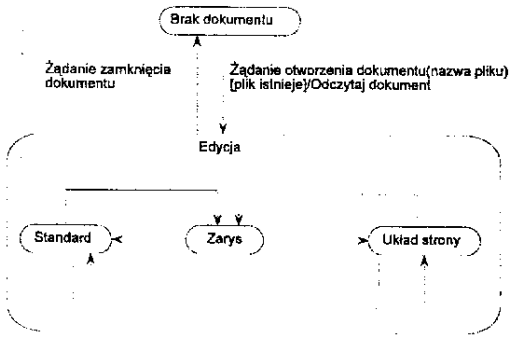
W przypadku systemów (fragmentów systemu) o złożonej dynamice, diagram przejść stanów mógłby być bardzo złożony i trudny do budowy. Rozwiązaniem tego problemu może być budowa hierarchii diagramów. Akcje i operacje, które przy pewnym poziomie ogólności zostały uznane za elementarne, mogą przy bardziej szczegółowej analizie okazać się złożone. Takie akcje i operacje można dekomponować do pełnego diagramu przejść stanów. Przykład takiej dekompozycji przedstawiony jest na rysunku 5.26.

diagramie przedstawionym na rysunku znajdują się przejścia nie opisane przez żadarzenia. Oznacza to, że przejścia te powodowane są przez zdarzenia zakończenia cji wykonywanych w stanach, z których następuje przejście. Należy też zwrócić na to, że stany na diagramie niższego poziomu odpowiadają kolejno wykonywaperacjom składającym się na operację drukowania. Jest to jedyny sposób przedia na diagramie przejść stanów tego typu sekwencji operacji.



Rysunek 5.26. Przykład hierarchii diagramów przejść stanów

Pewne stany mogą być do siebie podobne. System (fragment systemu) przebywając w tych stanach może np. reagować w ten sam sposób na niektóre zdarzenia. Takie stany można uogólnić do stanu będącego generalizacją (uogólnieniem) tych stanów. Edytor za pomocą którego pisana jest ta książka pozwala np. na edycję dokumentu w trzech różnych trybach. Tryby te różnią się sposobem reakcji na pewne zdarzenia. Większość zdarzeń jest jednak obsługiwana w ten sam sposób. Można więc te stany uogólnić wprowadzając stan Edycja (porównaj rysunek 5.27).



Rysunek 5.27. Przykład uogólnienia stanów

5.3.4. Specyfikacja modelu obiektowego

Diagramy obiektowe są oczywiście uzupełniane dodatkową informacją — specyfik... Specyfikacja służy przede wszystkim do bardziej szczegółowego opisu elementów p... wiających się na diagramach graficznych, chociaż pewne elementy specyfikacji nie... ją swoich bezpośrednich odpowiedników na diagramach.

Typowymi składowymi specyfikacji, stosowanymi do opisu wszystkich elementów są: nazwa oraz ogólny opis. Dodatkowe składowe specyfikacji klas to:

- lista pól,
- lista metod,
- ograniczenia, jakie muszą spełniać pola tej klasy (także w powiązaniu z polami innych klas),
- szacowana lub dokładna liczba obiektów tej klasy,
- trwałość, czyli informacja o tym czy obiekty danej klasy są tworzone tylko chwilowo, na czas uruchamiania programu, czy też muszą być przechowywane także w momencie, gdy program nie pracuje.

Metody specyfikuje się podając następujące informacje:

- dane wejściowe,
- dane wyjściowe,
- algorytm,
- warunki wstępne, tj. warunki jakie muszą spełniać pola klasy, do której należy metoda (a także pola innych klas) oraz dane wejściowe, aby metoda mogła rozpocząć się poprawnie,
- warunki końcowe, tj. warunki jakie muszą spełniać pola klasy, do której należy metoda (a także pola innych klas) oraz dane wyjściowe po zakończeniu poprawnie rozpoczętej oraz poprawnie wykonanej metody,
- wyjątki, tj. sytuacje wyjątkowe, które mogą zajść w trakcie realizacji metody (np. błąd odczytu pliku),
- złożoność czasowa,
- złożoność pamięciowa.

Na etapie analizy można nie opisywać dokładnie algorytmu metody odkładając tę czynność do fazy projektowania lub nawet implementacji. W przypadku prostych metod nazwa lub jednozdaniowy opis są wystarczające dla zrozumienia sposobu działania metody. Jeżeli jednak algorytm metody jest skomplikowany lub podobny efekt można osiągnąć stosują kilka algorytmów, przy czym analityk widzi korzyści płynące z wyboru jednego z nich, niezbędne staje się opisanie algorytmu metody.

Algorytmy można opisywać stosując język naturalny, schematy blokowe lub minispecyfikacje (ang. minispecification). Najczęściej stosowany jest ten ostatni sposób. Minispecyfikacje zapisuje się stosując tzw. pseudokod. Jest to zapis oparty na pewnym języku programowania, np. Pascalu, który posługuje się typowymi instrukcjami tego języka, łamie jednak szereg szczegółowych wymogów oraz zawiera fragmenty pisane językiem

naturalnym. Celem jest osiągnięcie precyzji opisu algorytmów typowej dla języków programowania przy większej czytelności. Poniżej podane są przykłady algorytmów zapisanych w formie minispecyfikacji oraz przykładowe realizacje w Pascalu.

Pierwszy przykład dotyczy algorytmu bisekcji (połowienia) służącego do znajdowania miejsca zerowego funkcji jednej zmiennej. Daną wejściową dla tego algorytmu jest początkowy przedział. Stosowany jest pseudokod wykorzystujący instrukcje podobne do stosowanych w Pascalu, w spolszczonej wersji.

Algorytm bisekcji — minispecyfikacja

jeżeli na jednym z końców przedziału funkcja osiąga wartość wystarczająco bliską zeru, **to**
    znalezione zostało rozwiązanie
w przeciwnym wypadku
jeżeli na końcach przedziału funkcja nie ma różnego znaku, **to**
    w podanym przedziale nie ma miejsca zerowego
w przeciwnym wypadku
powtarzaj
    podziel przedział na pół
    jeżeli w punkcie środkowym funkcja osiąga wartość wystarczająco bliską zeru, **to**
        znalezione zostało rozwiązanie
    w przeciwnym wypadku
    jeżeli przedział jest wystarczająco mały, **to**
        rozwiązaniem jest punkt środkowy przedziału
    w przeciwnym wypadku
        wybierz połowę przedziału, w której następuje zmiana znaku
dopóki nie zostanie znalezione rozwiązanie

Algorytm bisekcji — realizacja w Pascalu

```
procedure Bisekcja (x1, x2 : real; var bZnalezione :
Boolean; var x0 : real);
var x3 : real;
begin
   if abs(f(x1)) < rPROG1 then
   begin
        bZnalezione := true;
        x0 := x1
   end
   else
   if abs(f(x2)) < rPROG2 then
   begin
        bZnalezione := true;
        x0 := x2
   end
   else
   if f(x1)*f(x2) > 0 then
        bZnalezione := false
   else
   begin
        bZnalezione := false;
        repeat
             x3 := (x1+x2)/2;
             if abs(f(x3)) < rPROG1 then
             begin
                  bZnalezione := true;
                  x0 := x3
             end
             else
             if abs(x2-x1) < rPROG2 then
             begin
                  bZnalezione := true;
                  x0 := x3
             end
             else
             if f(x1)*f(x3) < 0 then
                  x2 := x3
             else
                  x1 := x3
        until bZnalezione
   end;
```


Kolejny przykład to algorytm z zakresu sztucznej inteligencji, klasyfikacji na podstawie reguł decyzyjnych. Reguła decyzyjna składa się z części warunkowej i decyzji. Reguła oznacza, że jeśli spełnione są warunki wymienione w części warunkowej, to należy podjąć podaną decyzję. Przykłady reguł to:

- jeżeli pacjent ma następujące objawy ..., to jest chory na ...,
- jeżeli wnioskodawca spełnia następujące warunki ..., to można mu udzielić kredytu.

Celem podanego algorytmu jest zaklasyfikowanie (czyli wybór właściwej decyzji) pewnego obiektu (np. pacjenta, wnioskodawcy o kredyt). Zakłada się, że reguły decyzyjne są uporządkowane od najważniejszej do najmniej ważnej.

Klasyfikacji — minispecyfikacja

dla od reguły najważniejszej do najmniej ważnej
    jeżeli obiekt spełnia warunki reguły, **to**
        podejmowana jest decyzja wskazywana przez regułę
dopóki nie podjęto decyzji lub nie sprawdzono wszystkich reguł

jeżeli nie podjęto decyzji, to
    podejmij decyzję wskazywaną przez regułę domyślną

Algorytm klasyfikacji — realizacja w Pascalu

```
function Klasyfikuj (Obiekt : PObiekt) : TDecyzja;
var
   bDecyzjaPodjeta : Boolean;
   i : integer;
begin
   bDecyzjaPodjeta := false;
   i := 1;
   repeat
        if Reguly [i].Dopasuj (Obiekt) then
        begin
             Klasyfikuj := Reguly [i].PobierzDecyzje;
             bDecyzjaPodjeta := TRUE
        end
        i := i + 1
   until bDecyzjaPodjeta or (i >= LiczbaRegul);
   if not bDecyzjaPodjeta then
        Klasyfikuj := RegulaDomyslna.PobierzDecyzje
end;
```

Pola elementarne, tj. pola nie składające się z prostszych elementów, specyfikuje się p
dając następujące informacje:

- typ przechowywanych wartości,
- jednostka miary,
- zakres dopuszczalnych wartości,
- lista możliwych wartości,
- wymagana precyzja,
- wartość domyślna,
- informacja o tym czy pole może być puste (nie posiadać wartości).

Dodatkowo dla wszystkich rodzajów pól można specyfikować:

- ograniczenia,
- metody, które mogą czytać, ustawiać i modyfikować wartości tego pola.

Pola nie muszą być danymi elementarnymi. Do opisu pól złożonych stosuje się sp
ne wyrażenia o podanej poniżej składni. Operandami wyrażenia opisującego z
strukturę danych mogą być zarówno dane elementarne, jak i kolejne złożone su
danych. Opisywane są one za pomocą identyfikatorów.

Operatorami używanymi w takich wyrażeniach są:

+	operator i,
(...)	operator występowania opcjonalnego,
[...|...]	operator wyboru jednej z opcji,
{...}	operator powtarzania,
m{...}n	operator powtarzania elementów od *m* do *n*,

Dane złożone pojawiające się opisie pól złożonych są opisywane za pomocą takich samych wyrażeń. Dane elementarne są opisywane tak samo jak pola elementarne. Dane złożone i elementarne są elementami specyfikacji nie mającymi bezpośredniego odpowiednika na diagramach graficznych.

Poniżej podane są przykłady zastosowania powyższej notacji.

Dane osobowe
    Imię +
    Nazwisko +
    Adres

Dane osobowe składają z imienia, nazwiska i adresu. Imię i nazwisko to dane elementarne. Adres to dana złożona zdefiniowana poniżej.

Adres
    Ulica +
    Numer domu +
    (Numer mieszkania) +
    Miasto +
    Kod

Numer mieszkania jest dana opcjonalną, która nie musi pojawiać się zawsze.

Linia
    {Punkt}

Linia składa się z wielu punktów.

Punkt
    Współrzędna pozioma +
    Współrzędna pionowa

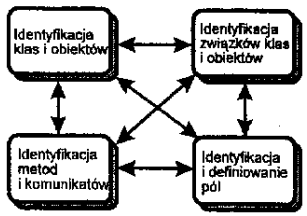
5.4. Proces tworzenia modelu obiektowego

W procesie budowy modelu obiektowego można wyróżnić cztery główne zadania:

- identyfikację klas i obiektów,
- identyfikację związków klas i obiektów,
- identyfikację i definiowanie pól,
- identyfikację i definiowanie metod i komunikatów.

Trzy pierwsze zadania służą budowie statycznego modelu klas. Celem ostatniego zadania jest zamodelowanie sposobu w jaki obiekty współpracując ze sobą wykonują funkcje systemu.

Często podkreśla się, że budowa modelu obiektowego nie ma charakteru liniowego a poszczególne zadania wykonywane są iteracyjnie. Z drugiej strony z reguły istnieje jedn pewien ogólny schemat, którym posługuje się analityk. Jednym z typowych podejść j rozpoczęcie analizy obiektowej od budowy statycznego modelu klas. W ramach tej czy ności najpierw następuje identyfikacja klas i obiektów, potem związków klas i obiekt a następnie identyfikacja pól. Następnie poszukuje się metod i komunikatów. Ta ogó kolejność zachowana jest w niniejszej książce. Nie ma to być jednak zachętą do ścisł zachowywania tego właśnie schematu. Czytelnik zauważy, że podczas omawiania cz ności, które zgodnie z tym schematem są wcześniejsze, w pewnych momen przydatne okażą się wyniki zadań późniejszych. Przykładowo znajomość pól i me klas jest pomocna przy określaniu ich związków. W praktyce analityk może więc od od tego schematu wykonując w danym momencie czynności, które na tym etapie u za najbardziej przydatne.



Rysunek 5.28. Proces budowy modelu obiektowego

Inny ogólny schemat realizacji procesu budowy modelu obiektowego polega na częciu od analizy sposobu realizacji poszczególnych funkcji systemu. Tworząc riusz, który opisuje sposób realizacji danej funkcji analityk jednocześnie ident klasy i ich metody. Dopiero potem następuje identyfikacja związków klas i ob oraz identyfikacja pól.

5.4.1. Budowa statycznego modelu klas

5.4.1.1. Identyfikacja klas i obiektów

Poniżej zostanie omówionych kilka sposobów identyfikowania klas i obiektów. Jak już wspomniano wcześniej nie są to ścisłe algorytmy i reguły, lecz raczej zbiory ogólnych wskazówek ułatwiających wykonanie tej czynności. Należy jednak podkreślić duże znaczenie doświadczenia analityka. Doświadczony analityk potrafi często z dużą łatwością zidentyfikować sensowne klasy i obiekty bez jawnego stosowania żadnej z wymienionych poniżej metod.

Podejście klasyczne

Klasyczne podejście do problemu poszukiwania klas obiektów opiera się na obserwacji, że w wielu systemach pojawiają się podobne klasy i obiekty. Proponuje się wobec tego listy typowych klas. Analityk przeglądając taką listę może poszukiwać podobnych klas w analizowanym systemie. Takie typowe klasy to:

- przedmioty namacalne (samochód, czujnik),
- role pełnione przez osoby (pracownik, wykładowca, polityk),
- zdarzenia, o których system przechowuje informacje (lądowanie samolotu, zamówienie, dostawa),
- interakcje pomiędzy osobami i/lub systemami, o których system przechowuje informacje (pożyczka, spotkanie, konferencja),
- lokalizacje — miejsca przeznaczone dla ludzi lub przedmiotów,
- grupy przedmiotów namacalnych (samochody, czujniki),
- organizacje (firma, wydział, związek),
- koncepcje (miara jakości, zadanie),
- dokumenty (prawo jazdy, faktura),
- klasy będące interfejsami dla systemów zewnętrznych,
- klasy będące interfejsami dla urządzeń sprzętowych.

Należy podkreślić, że elementami systemu są często tak zwane metadane, czyli opisy osób, przedmiotów namacalnych itp. Klasy będące fragmentami systemu opisują więc byty znajdujące się poza systemem. Ściśle rzecz biorąc do nazwy takich klas należałoby zawsze dodawać słowo „opis" (lub inne o podobnym znaczeniu), na przykład Opis pracownika, Dane samochodu. Analitycy często z tego rezygnują, chociaż może to być źródłem nieporozumień. Pracownik może na przykład oznaczać zarówno klasę, jak i system zewnętrzny — użytkownika systemu.

Analiza dziedziny problemu (ang. *problem domain analysis*)

Podejście to opiera się na obserwacji, że wiedza ekspertów i zawarta w literaturze na temat danej dziedziny często zawiera praktycznie gotowy model, który musi zostać jedynie zapisany w odpowiedniej formie. Poszukiwanie klas i obiektów odbywa się więc w ścisłej współpracy z ekspertami z danej dziedziny. Wartościowym źródłem informacji jest też specjalistyczna literatura, udział w szkoleniach, seminariach i prezentacjach. Należy zwrócić uwagę jakich bytów i klas bytów dotyczy wiedza na temat dziedziny problemu. Często podkreśla się przy tym rolę rysunków pojawiających się w pozycjach literaturowych i wykorzystywanych w czasie prezentacji. Coś co daje się narysować jest potencjalnie dobrą klasą.

Analiza dziedziny problemu obejmuje też wykorzystanie wyników poprzednich analiz podobnych systemów.

Analiza opisu w języku naturalnym

Podejście to rozpoczyna się od wykonania — przez eksperta z danej dziedziny — krótkiego opisu modelowanego (fragmentu) systemu w języku naturalnym. W opisie tym wyróżnia się następnie rzeczowniki (wraz z opisującymi je przymiotnikami) oraz czasowniki. Rzeczowniki traktuje się jako potencjalne klasy, obiekty lub pola. Czasowniki to potencjalne metody lub związki klas.

Ze względu na dużą elastyczność i niejednoznaczność języka naturalnego zapis taki powinien być jednak poddany uważnej analizie. Należy zwrócić uwagę na rzeczowniki odczasownikowe (wykonywanie — wykonywać). Należy też rozważyć czy różne rzeczowniki nie opisują w gruncie rzeczy tych samych bytów, a różne czasowniki tych samych czynności. Pewne rzeczowniki, takie jak: informacje, dane, opis, są z reguły zbyt ogólne aby pozwalały wprowadzić nowe klasy. Należy raczej zwrócić uwagę na to czego dotyczą informacje, dane, opis.

Metoda ta, choć krytykowana za swą nieprecyzyjność, pozwala z reguły stworzyć dobry szkielet modelu klas. Nie należy jednak spodziewać się, że pozwoli wykryć wszystkie klasy, związki, pola i metody. Metodę tę można szczególnie zalecić analitykom przystępującym do analizy bardzo słabo znanej dziedziny problemu.

Wykorzystanie związków klas i obiektów

Metoda ta wykorzystuje wyniki czynności, która nie została jeszcze omówiona. Jest więc przykład na to, że proces analizy obiektowej nie jest w pełni sekwencyjny.

W pewnych przypadkach analityk może zauważyć fakt występowania pewnego związku przed wprowadzeniem wszystkich klas, które się w tym związku pojawią. Po zidentyfikowaniu klasy Pracownik, analityk może zauważyć, że pracownicy pozostają w związku zatrudnienia, należy więc wprowadzić dodatkową klasę, na przykład Przedsiębiorstwo.

Zgodnie z tą metodą dla danej klasy należy rozważyć:

- Czy ma ona potencjalne specjalizacje i/lub generalizacje?
- Czy ma części składowe i/lub jest częścią większej całości?
- Czy pozostaje w związkach z innymi klasami?

Analiza funkcji (przypadków użycia)

Opisane dotąd metody koncentrowały się na opisie fragmentu dziedziny problemu. Nie wykorzystywały one informacji o funkcjach wykonywanych przez system. Mogą więc prowadzić do wykrycia wielu klas, które są co prawda sensowne w ramach dziedziny problemu, nie leżą jednak w zakresie odpowiedzialności systemu.

Metoda ta rozpoczyna się od wyboru pewnej funkcji systemu (niekoniecznie elementarnej). Następnie w języku naturalnym opisuje się sposób w jaki system wykonuje tę funkcję. Na tej podstawie tworzy się scenariusz interakcji obiektów jednocześnie wprowadzając niezbędne klasy i metody. Analiza scenariuszy zostanie omówiona dokładniej w dalszej części książki, z naciskiem na jej wykorzystanie dla identyfikacji metod.

Weryfikacja klas i obiektów

Opisane powyżej metody mogą prowadzić do wykrycia znacznej liczby potencjalnych klas, które nie znajdą się jednak w ostatecznym modelu. Niektórzy autorzy wręcz zalecają rozpoczęcie analizy od zidentyfikowania jak największej liczby potencjalnych klas, a następnie odrzucanie niepotrzebnych klas. Weryfikując konieczność wprowadzenia danej klasy należy wziąć pod uwagę następujące czynniki:

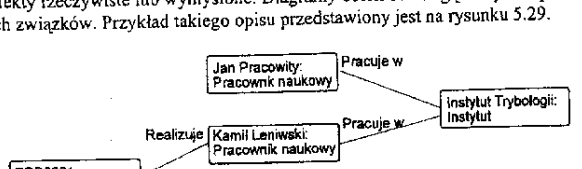
- Nieobecność pól i metod — z reguły oznacza to, że klasa znajduje się poza zakresem odpowiedzialności systemu.
- Nieliczne (pojedyncze) pola i metody — jest to sugestia, że te pola i metody można by umieścić w innych klasach.
- Tylko jeden obiekt w klasie — często oznacza to zbyt rozbudowaną hierarchię specjalizacji. **Dobrą klasą** jest klasa Samochód, złymi Samochód Kowalskiego i Samochód Nowaka.
- Brak związków z innymi klasami — z reguły oznacza to, że klasa znajduje się poza zakresem odpowiedzialności systemu.

Należy zauważyć, że nie istnieją jednoznaczne reguły pozwalające stwierdzić kiedy należy wprowadzić pole, a kiedy odrębną klasę. Każde pole może być zrealizowane jako odrębna klasa powiązana związkiem agregacji. Każda klasa, której obiekty są składowymi obiektów innej klasy, może zostać wprowadzona jako pole(a) do klasy będącej całością. W pewnych systemach obiektowych, na przykład języku programowania Smalltalk, każde pole jest także obiektem. Z drugiej strony każdy system można zrealizować tak, aby zawierał wyłącznie jeden globalny obiekt przechowujący wszystkie dane w swoich polach i realizujący wszystkie funkcje za pomocą swoich metod. Byłoby to równoważne z realizacją projektu strukturalnego. W praktyce rozróżnienie pól i klas zależy więc od subiektywnej opinii danego analityka.

5.4.1.2. Identyfikacja związków klas i obiektów

Związki klas

Związek klas jest uogólnionym opisem wielu pojedynczych związków pojawiających się pomiędzy poszczególnymi obiektami tych klas. Przy modelowaniu tych związków przydatny może być więc opis kilku konkretnych związków pomiędzy obiektami. Mogą to być obiekty rzeczywiste lub wymyślone. Diagramy obiektów mogą służyć do prezentacji takich związków. Przykład takiego opisu przedstawiony jest na rysunku 5.29.



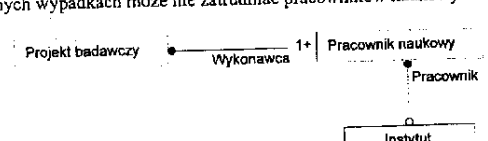
Rysunek 5.29. *Diagram obiektów przedstawiający kilka konkretnych związków pomiędzy obiektami*

Ponieważ diagramy takie mają tylko chwilowe zastosowanie analitycy posługują się odręcznymi rysunkami. Tworząc taki diagram należy przede wszystkim rozważyć:

- Czy może istnieć obiekt nie pozostający w danym związku? Czy na przykład pracownik naukowy może nie realizować żadnego projektu badawczego?
- Czy może istnieć obiekt pozostający w danym związku z wieloma obiektami? Czy na przykład pracownik naukowy może realizować kilka projektów badawczych?

Szczególnie ważne jest pierwsze z tych pytań. Może się wydawać, że instytut wyższej uczelni zawsze zatrudnia pracowników naukowych. Może to jednak nie być prawdą w przypadku nowo utworzonego instytutu.

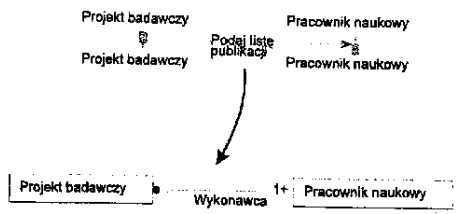
Konkretne związki przedstawione na rysunku 5.29 mogą zostać uogólnione na diagramie klas w sposób przedstawiony na rysunku 5.30. Wzięto tu pod uwagę fakt, że instytut w szczególnych wypadkach może nie zatrudniać pracowników naukowych.



Rysunek 5.30. *Związki klas wprowadzone na podstawie diagramu z rysunku 5.29*

Czasowniki pojawiające się w słownym opisie (fragmentu) systemu mogą oznaczać związki klas. Przykładowo zwrot „pracownik naukowy realizuje projekt badawczy" sugeruje istnienie związku pomiędzy klasami Pracownik naukowy i Projekt badawczy.

Fakt wysyłania komunikatów przez obiekty pewnej klasy do obiektów innej klasy sugeruje istnienie związku między tymi klasami (porównaj rysunek 5.31). Nie jest to jednak reguła jednoznaczna. Obiekt może wysyłać komunikat do obiektu klasy, z którą związek jest pośredni. Pewne klasy mogą z kolei mieć wyłącznie pojedyncze obiekty, dostępne globalnie dla wszystkich obiektów w systemie.

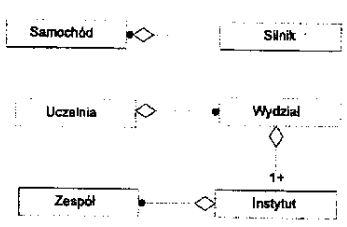


Rysunek 5.31. *Fakt wysyłania komunikatu wskazujący na związek klas*

Związki agregacji

Na występowanie związków agregacji wskazują takie zwroty pojawiające się w słownym opisie systemu jak: zawiera, składa się, obejmuje.

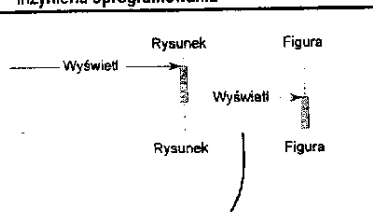
Poszukując związków agregacji należy zwracać uwagę na klasy posiadające części składowe i klasy będące zbiorami pewnych elementów. Należy też rozważyć dla poszczególnych klas czy nie wchodzą one w skład większej całości lub zbioru (porównaj rysunki 5.32 i 5.33).



Rysunek 5.32. *Samochód składa się (między innymi) z silnika. Uczelnia składa się z wydziałów. Wydział składa się z instytutów. Instytut składa się z zespołów*

Rysunek 5.33. *Grupa studencka jest zbiorem studentów. Zbiór podatników jest opisywany przez klasę Podatnicy*

Czynnikiem wskazującym na występowanie związku agregacji jest też propagacja komunikatów, czyli sytuacja, w której obiekt pewnej klasy otrzymujący komunikat wysyła taki sam komunikat(y) do obiektów innych klas (porównaj rysunek 5.34).



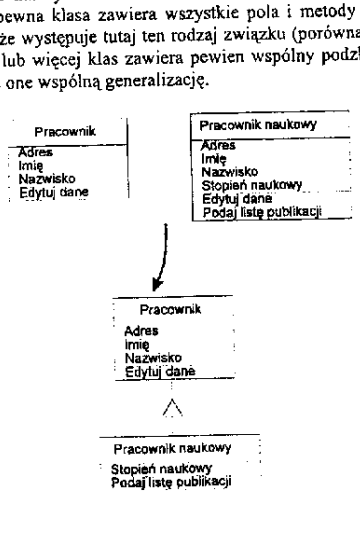
Rysunek 5.34. Propagacja komunikatów wskazująca na związek agregacji

Każdy związek agregacji może zostać zamodelowany jako związek klas o nazwie „składa się" lub podobnej. W przypadku pewnych środowisk programistycznych, na przykład relacyjnych baz danych, nie ma też żadnej różnicy w realizacji tych dwóch rodzajów związków. Analityk znający tę cechę środowiska implementacji może więc zrezygnować z wyróżniania związku agregacji.

Związki generalizacji-specjalizacji

Poszukując związków generalizacji-specjalizacji należy zwrócić uwagę na istniejące w danej dziedzinie problemu klasyfikacje. Klasyfikacja wyrobów pewnej firmy może przykładowo sugerować możliwość wprowadzenia odpowiednich związków generalizacji-specjalizacji.

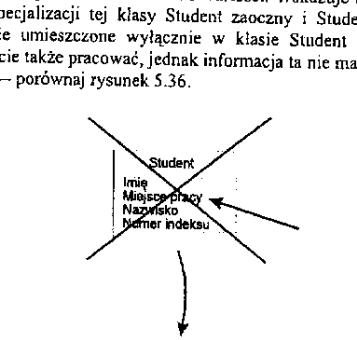
Jak wspomniano w sekcji 5.3.1, omawiającym diagramy klas i obiektów, związku generalizacji-specjalizacji nie należy utożsamiać z dziedziczeniem pól i metod. Z drugiej jednak strony fakt, że pewna klasa zawiera wszystkie pola i metody innej klasy jest silną wskazówką na to, że występuje tutaj ten rodzaj związku (porównaj rysunek 5.35). Podobnie fakt, że dwie lub więcej klas zawiera pewien wspólny podzbiór pól i metod wskazuje, że mogą mieć one wspólną generalizację.



Rysunek 5.35. Dziedziczenie pól i metod często wskazuje na występowanie związku generalizacji-specjalizacji

Nazwa specjalizacji często składa się z nazwy generalizacji z dodatkowym przymiotnikiem (przymiotnikami). Nazwa klasy Pracownik naukowy wskazuje więc, że jest to specjalizacja klasy Pracownik. Nazwa klasy Samochód osobowy wskazuje, że jest to specjalizacja klasy Samochód. Wyróżnienie klas o nazwach zawierających ten sam fragment z różnymi przymiotnikami sugeruje, że klasy te można uogólnić. Przykładowo nazwy klas Pracownik naukowy i Pracownik techniczny sugerują możliwość wprowadzenia ogólniejszej klasy Pracownik.

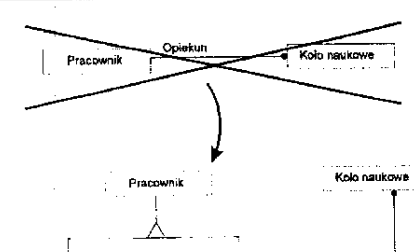
Na to, że dana klasa może mieć specjalizacje wskazuje też fakt, że pewne jej pola nie mają sensownych wartości dla niektórych obiektów. Na przykład pole Miejsce pracy klasy Student nie zawsze będzie miało sensowne wartości. Wskazuje to na konieczność wyróżnienia dwóch specjalizacji tej klasy Student zaoczny i Student dzienny. Pole Miejsce pracy zostanie umieszczone wyłącznie w klasie Student zaoczny (student dzienny może oczywiście także pracować, jednak informacja ta nie ma znaczenia z punktu widzenia uczelni) — porównaj rysunek 5.36.



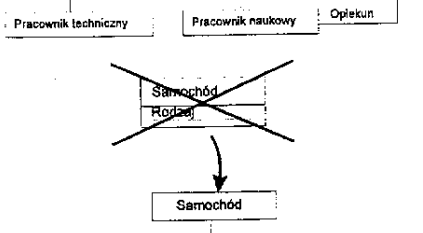
Rysunek 5.36. Pole, które nie zawsze ma sensowne wartości wskazuje na konieczność wprowadzenia specjalizacji

Również metody, które nie mają sensu dla pewnych obiektów mogą wskazywać na konieczność wprowadzenia specjalizacji. W podobny sposób należy rozważyć związki, w których biorą udział obiekty danej klasy. Mogą one mieć sens tylko dla pewnych jej specjalizacji (porównaj rysunek 5.37).

Na możliwość wprowadzenia specjalizacji wskazują też pola, które przyjmują jedynie kilka powtarzających się wartości. Pola takie często błędnie wprowadza się dla rozróżnienia różnych typów (specjalizacji) danej klasy. Pole Rodzaj klasy Samochód przyjmujące jedynie wartości „osobowy" i „dostawczy" wskazuje np. na konieczność wprowadzenia specjalizacji tej klasy (porównaj rysunek 5.38).



Rysunek 5.37. *Związek, który ma sens tylko dla pewnych obiektów wskazuje na konieczność wprowadzenia specjalizacji*



Rysunek 5.38. *Pole o nielicznych, powtarzających się wartościach wskazujące na konieczność wprowadzenia specjalizacji*

Weryfikacja związków

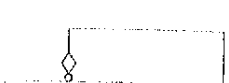
Do weryfikacji poprawności związków w dziedzinie problemu służy tzw. test semantyczny. Dla danych dwóch klas A i B, dla których wprowadzono pewien związek, należy odpowiedzieć na pytania:

- Czy w ramach danej dziedziny problemu sensownie brzmi zdanie „A jest rodzajem B" (jeżeli klasa A jest specjalizacją klasy B)?
- Czy w ramach danej dziedziny problemu sensownie brzmi zdanie „A [czasownik] B" (jeżeli klasy A i B są związane związkiem klas)?
- Czy w ramach danej dziedziny problemu sensownie brzmią zdania „A jest częścią B" lub „B składa się (zawiera) A" (jeżeli obiekty klasy A są składowymi obiektów klasy B)?

Szczególną uwagę należy zwrócić na związki obligatoryjne, to jest związki o krotności większej lub równej 1. Krotność taka oznacza, że każdy obiekt danej klasy musi pozostawać w co najmniej jednym związku tego typu. Często po dokładniejszej analizie (lub co gorsza w trakcie eksploatacji systemu) okazuje się, że w wyjątkowych sytuacjach pewne obiekty mogą nie brać udziału w takim związku.

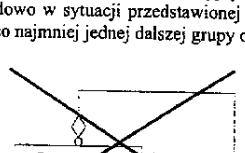
Cykle nie są dopuszczalne w przypadku związków generalizacji-specjalizacji. Klasa nie może być jednocześnie swoją specjalizacją i generalizacją. W przypadku związków

agregacji cykle są dopuszczalne, jednak w praktyce pojawiają się bardzo rzadko (porównaj rysunek 5.39).



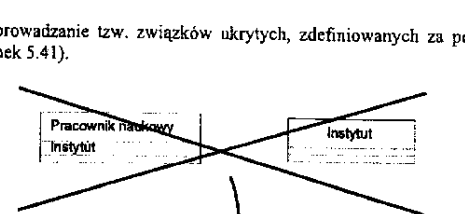
Rysunek 5.39. *Cykle są dopuszczalne w związkach agregacji*

Cykle nie mogą oczywiście zawierać związków obligatoryjnych, gdyż prowadzi to do nieskończonych pętli. Przykładowo w sytuacji przedstawionej na rysunku 5.40, każda grupa osób musi składać się z co najmniej jednej dalszej grupy osób.



Rysunek 5.40. *Cykl zawierający związek obligatoryjny prowadzi do nieskończonej pętli*

Błędem jest wprowadzanie tzw. związków ukrytych, zdefiniowanych za pomocą pól (porównaj rysunek 5.41).



Rysunek 5.41. *Związek ukryty*

Ponieważ związki klas mogą być nazywane w różny sposób, a te same klasy mogą być umieszczane na różnych diagramach istnieje ryzyko wielokrotnego wprowadzenia nadmiarowych związków opisujących dokładnie ten sam fakt.

5.4.1.3. Identyfikacja i definiowanie pól

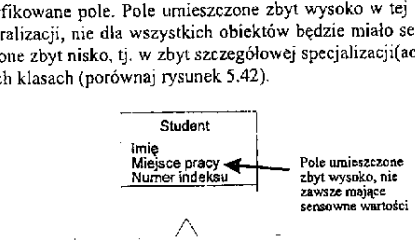
Identyfikując pola klas należy odpowiedzieć na następujące pytania:

- Co jest potrzebne do opisu danej klasy w ramach dziedziny problemu?
- Jakie dane będą potrzebne metodom danej klasy do realizacji ich zadań?
- Jakie pola należy wprowadzić, aby opisać stany w jakich mogą znajdować się obiekty danej klasy?

Opis (fragmentu) systemu w języku naturalnym może być pomocny podczas realizacji tego zadania. Jak wspomniano już wcześniej, rzeczowniki znajdujące się w takim opisie mogą odpowiadać zarówno klasom, jaki i polom klas. Tego typu słowny opis rzadko jest jednak na tyle szczegółowy, aby pozwolił zidentyfikować wszystkie pola.

Zaakceptowany przez klienta prototyp interfejsu użytkownika dostarcza istotnych informacji o wprowadzanych, edytowanych i prezentowanych użytkownikowi danych. Dane te odpowiadają polom klas systemu.

Czasami nie jest jasne na jakim poziomie w hierarchii generalizacji-specjalizacji należy umieścić już zidentyfikowane pole. Pole umieszczone zbyt wysoko w tej hierarchii, tj. w zbyt ogólnej generalizacji, nie dla wszystkich obiektów będzie miało sensowne wartości. Pole umieszczone zbyt nisko, tj. w zbyt szczegółowej specjalizacji(ach), może powtarzać się w różnych klasach (porównaj rysunek 5.42).



Rysunek 5.42. Błędy w umieszczaniu pól w hierarchii generalizacji-specjalizacji

Po zidentyfikowaniu pól zadaniem analityka jest ich szczegółowa specyfikacja w sposób omówiony w sekcji 5.3.4 omawiającej specyfikację modelu obiektowego.

5.4.2. Identyfikacja i definiowanie metod i komunikatów

Metody można podzielić na dwie grupy:

- algorytmicznie proste,
- algorytmicznie złożone.

Metody algorytmicznie proste to:

- Konstruktory, czyli metody tworzące nowe obiekty danej klasy. W tej książce przyjęta jest konwencja stosowana między innymi w języku C++, zgodnie z którą metoda o nazwie takiej samej jak nazwa klasy jest konstruktorem. Umieszczenie na diagramie interakcji komunikatu wywołującego konstruktor pozwala zobrazować fakt tworzenia nowego obiektu danej klasy.

- Destruktory, czyli metody usuwające obiekty danej klasy. W niniejszej książce przyjęta jest konwencja stosowana między innymi w języku C++, zgodnie z którą metoda o nazwie takiej samej jak nazwa klasy poprzedzona znakiem tyldy „~" jest destruktorem. Umieszczenie na diagramie interakcji komunikatu wywołującego destruktor pozwala zobrazować fakt usuwania obiektu danej klasy.
- Metody służące do pobierania/ustawiania wartości pól danej klasy. Często nazywa się je PobierzDane, UstawDane, PobierzPole, UstawPole (ang. *SetData, GetData, SetField, GetField*). Dwie pierwsze metody służą do ustawiania/pobierania wartości wszystkich pól, dwie dalsze do ustawiania/pobierania wartości pojedynczego pola. Warto zwrócić uwagę na fakt, że omawiany model obiektowy nie zakłada możliwości bezpośredniego dostępu metody do pól innych obiektów niż ten, na rzecz którego została wywołana mimo, że jest to możliwe w wielu językach obiektowych. W praktyce metody służące do pobierania/ustawiania wartości pól są często implementowane jako bezpośrednie odwołania do pól (z wyjątkiem sytuacji, w których niezbędne jest sprawdzanie dodatkowych warunków spójności podczas ustawiania wartości pól). Dlatego też metod tych często nie specyfikuje się na etapie analizy, gdyż sposób ich działania jest oczywisty.
- Metody służące do ustawiania związków pomiędzy obiektami.
- Metody służące do edycji pól danej klasy.

W wielu systemach metody algorytmicznie proste stanowią zdecydowaną większość wszystkich metod.

Metody algorytmicznie złożone to na przykład:

- metody służące do wykonywania obliczeń,
- metody śledzące (monitorujące) pracę urządzeń i systemów zewnętrznych.

5.4.2.1. Analiza scenariuszy (przypadków użycia)

Analiza scenariuszy polega na definiowaniu sposobu realizacji poszczególnych funkcji systemu. Zgodnie z tym podejściem analityk wybiera poszczególne funkcje systemu, a następnie modeluje przepływ komunikatów niezbędnych dla realizacji tych funkcji. Wybrana funkcja nie musi być funkcją elementarną. Często bardziej naturalne jest rozważanie funkcji wyższego poziomu, szczególnie wtedy, gdy funkcje składowe są ze sobą ściśle powiązane.

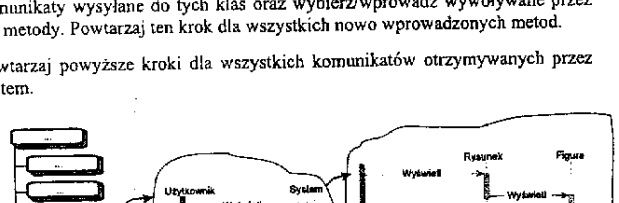
Jedna z proponowanych dróg postępowania polega na określeniu zewnętrznego scenariusza przepływu komunikatów pomiędzy modelowanym systemem a systemami zewnętrznymi zaangażowanymi w realizację tej funkcji. Jeżeli jedynym systemem zewnętrznym zaangażowanym w realizację funkcji jest użytkownik, scenariusz zewnętrzny opisuje komunikaty jakie użytkownik wysyła do modelowanego systemu, czyli polecenia jakie wydaje użytkownik. W budowie scenariusza zewnętrznego pomocne są

prototypy interfejsu użytkownika. Prototyp zawiera informacje o tym, jakie polecenia wydaje użytkownik korzystając z menu, pasków pomocy, przycisków w dialogach, przycisków myszy, poleceń wpisywanych z klawiatury. Polecenia te często odpowiadają elementarnym funkcjom systemu.

Pewne założenia dotyczące sposobu i środowiska realizacji interfejsu użytkownika są zresztą niezbędne dla określenia scenariuszy zewnętrznych. Jeżeli na przykład przewidywane środowisko implementacji zawiera funkcję służącą do edycji zbiorów elementów, umożliwiającą przeglądanie, dodawanie, usuwanie i edytowanie elementów (możliwości takie są typowe dla języków czwartej generacji), to edycja takiego zbioru wymaga od użytkownika wydania tylko jednego polecenia. Jeżeli środowisko implementacji nie dostarcza takich możliwości, niezbędne będzie wprowadzenie poleceń służących na przykład do dodawania lub usuwania elementów.

Kolejnym krokiem jest określenie scenariusza przepływu komunikatów między obiektami systemu. Scenariusz ten jest tworzony według następującego schematu (porównaj rysunek 5.43):

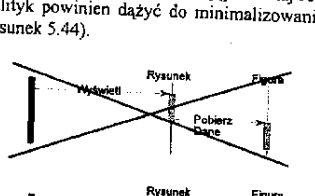
- Wybierz jeden z komunikatów otrzymywanych przez system.
- Określ klasę, której obiekt(y) otrzyma ten komunikat.
- Wybierz metodę, która będzie wywoływana przez ten komunikat lub wprowadź taką metodę jeżeli nie została jeszcze wprowadzona.
- Rozważ czy wprowadzona metoda jest elementarna, tj. czy może w pełni wykonać związane z otrzymanym komunikatem polecenie. Jeżeli metoda nie jest elementarną określ klasy, których obiekty będą uczestniczyły w jej realizacji. Dodaj komunikaty wysyłane do tych klas oraz wybierz/wprowadź wywoływane przez nie metody. Powtarzaj ten krok dla wszystkich nowo wprowadzonych metod.
- Powtarzaj powyższe kroki dla wszystkich komunikatów otrzymywanych przez system.



Rysunek 5.43. Analiza scenariuszy polegająca na transformacji: funkcja systemu → scenariusz zewnętrzny → scenariusz przepływu komunikatów

Jeśli zachodzi konieczność dodania metody może się okazać, że nie istnieje jeszcze klasa, w której można by te metodę umieścić. Analiza scenariuszy pozwala więc także wykryć dodatkowe klasy.

Wiele prostych funkcji, które często stanowią sporą część funkcji systemu, może zostać wykonanych przez pojedynczą metodę nie odwołującą się do danych przechowywanych w różnych obiektach. W przypadku bardziej złożonych funkcji konieczne jest korzystanie z danych przechowywanych przez różne obiekty. Tworząc scenariusz przepływu komunikatów należy metody umieszczać w klasach, w których umieszczone są pola potrzebne dla ich wykonania. Oczywiście zawsze możliwe jest zrealizowanie całego algorytmu scenariusza przez jedną metodę. W tym wypadku niezbędne staje się jednak wprowadzenie wielu komunikatów służących do pobierania danych z innych obiektów, tj. komunikatów wywołujących metody typu PobierzDane, PobierzPole. Prowadzi to do klas słabo spójnych oraz silnie powiązanych pomiędzy sobą (porównaj rozdział 6.7 omawiający jakość projektu). Analityk powinien dążyć do minimalizowania liczby tego typu komunikatów (porównaj rysunek 5.44).

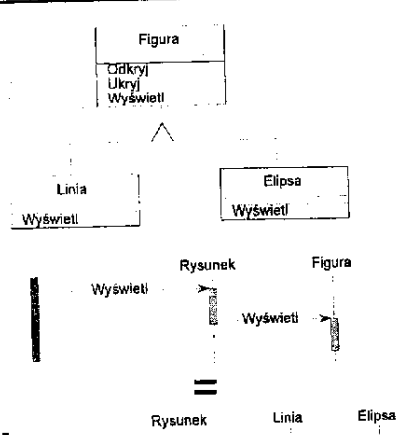


Rysunek 5.44. Metody powinny być umieszczane w klasach, w których umieszczone są wykorzystywane przez nie pola

Metody w modelu obiektowym są metodami wirtualnymi. Jeżeli klasa Figura posiadająca metodę Wyświetl ma specjalizacje, np. Linia, Elipsa, które redefiniują metodę wyświetl, to umieszczenie na diagramie interakcji komunikatu Wyświetl skierowanego do obiektu klasy Figura, oznacza wywołanie metody Wyświetl klas Linia lub Elipsa. Inaczej mówiąc z punktu widzenia klasy wysyłającej ten komunikat, nie ma znaczenia, czy obiekt, który ten komunikat odbierze jest obiektem klasy Figura lub jednej z jej specjalizacji. W każdym wypadku uruchomiona powinna zostać metoda właściwej klasy (porównaj rysunek 5.45).

Inne podejście do analizy scenariuszy rozpoczyna się od stworzenia słownego opisu sposobu w jaki system realizuje daną funkcję. Opis ten odpowiada skryptowi umieszczanemu na diagramie interakcji (porównaj rozdział 5.3.2 omawiający notację tych diagramów). Następnie śledząc ten opis wprowadza się komunikaty realizujące wymienianie w nim czynności. Możliwe jest także wykrycie dodatkowych klas. Ogólnie rzecz biorąc czasowniki znajdujące się w opisie scenariusza odpowiadają komunikatom i metodom, rzeczowniki klasom, polom i parametrom komunikatów. Podejście to może być stosowane także wtedy, gdy nie zidentyfikowano wcześniej klas systemu. Analiza scenariuszy staje się wtedy również podstawową technika identyfikowania klas (porównaj rozdział 5.4.1.1 omawiający identyfikację klas i obiektów).

Stosując opisane powyżej sposoby tworzy się scenariusze przepływu komunikatów dla wszystkich funkcji systemu. Analiza scenariuszy zapewnia więc realizację wszystkich wymagań funkcjonalnych.



Rysunek 5.45.
Metoda wirtualna

Analiza scenariuszy jest często wykonywana przyrostowo. Na wstępie wybiera się pewien podzbiór najważniejszych funkcji. Dla nich modeluje się scenariusze przepływu komunikatów. Scenariusze te są następnie implementowane. W kolejnej iteracji wybiera się dalsze podzbiory scenariuszy (rozdział 2.5 omawia realizację przyrostową).

5.4.2.2. Modelowanie przejść stanów

Omówiona poprzednio analiza scenariuszy jest podstawową technika służącą identyfikowaniu metod i komunikatów wykonywaną dla wszystkich funkcji systemu. Modelowanie przejść stanów stosuje się w celu opisania tylko niektórych fragmentów systemu, w których zmiana stanów i reakcje na zdarzenia ma istotne znaczenie. Metodę tę stosuje się do modelowania zachowania dynamicznego całego systemu, pojedynczej klasy lub grupy powiązanych klas, jak również do modelowania zachowania się systemu w trakcie realizacji pewnej funkcji.

Istnieją dwie podstawowe techniki modelowania przejść stanów:

- stanocentryczna (ang. *state-driven*),
- zdarzeniocentryczna (ang. *event-driven*).

Technika stanocentryczna polega na wykonaniu następujących czynności:

- określ pewien początkowy stan modelowanego fragmentu systemu,
- określ operacje wykonywane w tym stanie oraz akcje wykonywane w momencie wejścia oraz wyjścia z tego stanu,

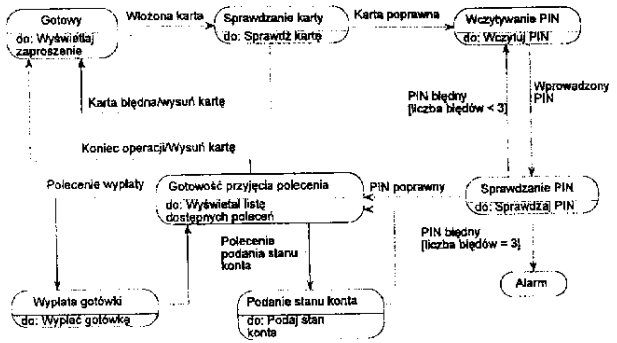
- określ zdarzenia mogące mieć wpływ na modelowany fragment systemu w tym stanie,
- dla każdego zdarzenia określ reakcję modelowanego fragmentu systemu na zajście tego zdarzenia, tj. akcję wykonywaną w momencie zajścia zdarzenia i ewentualne przejście do innego stanu,
- kontynuuj analizę dla każdego nie przeanalizowanego jeszcze stanu, do którego może nastąpić przejście.

Technika zdarzeniocentryczna stosowana jest dla modelowania sposobu realizacji przez system jego funkcji. Technika ta polega na wykonaniu następujących czynności:

- wybierz pewną, niekoniecznie elementarną, funkcję systemu,
- określ zdarzenia zewnętrzne jakie mogą zajść w trakcie realizacji tej funkcji,
- wybierz jedno ze zdarzeń zewnętrznych,
- określ akcję wykonywaną w momencie zajścia tego zdarzenia,
- określ stany na jakie może mieć wpływ to zdarzenie,
- dla każdego stanu określ reakcję na zajście rozważanego zdarzenia, czyli ewentualne przejście do innego stanu i akcje wykonywane w momencie jego zajścia,
- dla każdego wprowadzonego stanu określ wykonywane w nim operacje oraz akcje wykonywane w momencie wejścia oraz wyjścia z tego stanu,
- rozważ w powyższy sposób kolejne zdarzenia zewnętrzne,
- rozważ w podobny sposób zdarzenia wewnętrzne, które mogą się pojawić w trakcie wykonywania modelowanej funkcji, np. zdarzenia pojawiające się w momencie zakończenia wykonywania pewnej operacji.

W praktyce często nie stosuje się żadnej z tych technik w czystej postaci lecz raczej łączy się te dwa podejścia.

Model przejść stanów przedstawiony na rysunku 5.46 opisuje działanie bankomatu. Bankomat znajduje się na wstępie w stanie gotowości. Ze stanu tego może wyjść po włożeniu karty magnetycznej. Powoduje to przejście do stanu sprawdzania poprawności karty. Jeżeli karta jest niepoprawna (np. pochodzi z innego banku) jest ona wysuwana i następuje powrót do stanu gotowości. Jeżeli karta była poprawna następuje przejście do stanu wczytywania PIN, czyli numeru identyfikacyjnego. Po wprowadzeniu numer PIN podlega weryfikacji. Po dwóch pierwszych błędach klient może powtórzyć wprowadzanie numeru PIN. Po trzecim błędzie następuje przejście do stanu alarmu. Jeżeli PIN został podany poprawnie następuje przejście do stanu, w którym bankomat jest gotowy do przyjęcia polecenia. Klient może zażądać zakończenia operacji, co powoduje powrót do stanu gotowości. Wybór polecenia wypłaty powoduje przejście do stanu, w którym następuje wypłata gotówki (stan ten nie jest stanem elementarnym). Wybór polecenia podania stanu konta powoduje przejście do stanu, w którym podawany jest stan konta. Po wykonaniu poleceń następuje powrót do stanu gotowości przyjęcia zlecenia.



Rysunek 5.46. Przykład modelu przejść stanów

Modelu przejść stanów jest jedynie pośrednim wynikiem analizy. Końcowym celem jest budowa modelu klas, czyli identyfikacja i definicja klas, ich związków, pól i metod. Powstaje pytanie, w jaki sposób można wykorzystać model przejść stanów?

Akcje i operacje pojawiające się w tym modelu opisują czynności wykonywane przez metody klas należących do modelowanego fragmentu systemu. Dana akcja lub operacja nie musi jednak jednoznacznie odpowiadać pewnej metodzie. Może ona odpowiadać jedynie fragmentowi metody. Może też odpowiadać grupie metod, szczególnie w przypadku akcji lub operacji nieelementarnych.

Niektóre stany odpowiadają okresom, w których wykonywane są operacje (dotyczy to wszystkich stanów z rysunku 5.46). To, że modelowany fragment systemu znajduje się w danym stanie, wynika z faktu wykonywania odpowiedniej operacji. Stan, który nie odpowiada konkretnej operacji musi być identyfikowany w inny sposób. O tym, że modelowany fragment systemu znajduje się w danym stanie mogą informować wartości pól obiektów lub stan związków obiektów. Model przejść stanów pozwala więc wykryć pola, które należy wprowadzić dla identyfikowania stanów obiektów.

5.4.3. Przykłady

Metod analizy nie można się nauczyć bez ich praktycznego stosowania. Poniżej przedstawiono kilka przykładów zastosowania metod obiektowych do analizy wprowadzonych już wcześniej przykładowych systemów.

5.4.3.1. System podatkowy

Analiza tego systemu rozpocznie się od poszukiwania klas. Zastosowana zostanie analiza opisu w języku naturalnym. Jak wspomniano w sekcji 5.4.1.1 omawiającej metody identyfikacji klas i obiektów, jest to podejście stosowane głównie w przypadku (fragmentów) dziedzin problemu stanowiących nowość dla analityka.

Poniżej znajduje się opis systemu podatkowego w języku naturalnym, przedstawiony już w sekcji 4.2.1. Jeszcze raz należy przypomnieć, że jest to bardzo uproszczony opis rzeczywistej sytuacji. Dla ułatwienia analizy w opisie wytłuszczeniem wyróżniono rzeczowniki wraz z przymiotnikami, kursywą wyróżniono czasowniki.

System podatkowy — analiza opisu w języku naturalnym

Program *ułatwia* **przygotowywanie formularzy zeznań podatkowych** (PIT-ów) oraz **przechowywanie informacji** o **źródłach przychodów i ulg**. **Zeznanie** może *być* tworzone dla **pojedynczego podatnika** lub **małżeństwa**. **Zeznanie** *obejmuje* **informację** o **rocznych przychodach** (w przypadku **małżeństwa** z podziałem na **przychody męża i żony**) oraz o **ulgach podatkowych**. **Przychody** *podzielone* są na **klasy** ze względu na **źródło uzyskania**, na przykład **pozarolnicza działalność gospodarcza**, **wolny zawód**. W ramach danej **klasy przychodów podatnik** *mógł osiągnąć* **szereg przychodów z różnych źródeł** (jeżeli na przykład pracuje lub wykonuje zlecenia dla wielu firm). Wszystkie **przychody** *opisane* są przez **kwotę przychodu**, **kwotę kosztów**, **kwotę zapłaconych zaliczek** oraz **kwotę dochodu**. Powyższe **informacje** *pozwalają obliczyć* **należny podatek** oraz **kwotę do zapłaty/zwrotu**. **Zeznanie** *zawiera* także **informacje** o **podatniku(ach)** oraz **adres Urzędu Skarbowego**, do którego jest *kierowane*. **System** *pozwala wydrukować* **wzorzec zeznania** zawierający wszystkie **informacje** jakie **podatnik** *musi umieścić* w **formularzu**. **Zeznanie** można *zabezpieczyć* przed **dalszymi zmianami** (powinno to zostać zrobione po złożeniu zeznania w Urzędzie Skarbowym). **System** *pozwala* na **tworzenie listy podatników** oraz **Urzędów Skarbowych**, które *mogą być* pomocne podczas **tworzenia nowego zeznania**. *Przechowuje* także **informacje** o wszystkich **przygotowanych zeznaniach**.

Oczywiście nie można wszystkich rzeczowników w powyższym opisie uznać za potencjalne klasy. Analizując ten opis należy zwrócić uwagę na wymienione poniżej niejednoznaczności i niejasności.

Terminy: **formularz zeznania podatkowego, PIT**, **zeznanie** i **formularz** są używane zamiennie, opisują więc dokładnie ten sam byt. Również terminy **program** i **system** są używane zamiennie. Odnoszą się one do całości systemu. Podobnie ma się rzecz z terminami **źródło przychodów** i **źródło uzyskania.**

Rzeczownik **informacja** nie odpowiada żadnej odrębnej klasie. Zwrot **„informacje o źródłach przychodów i ulg"** wskazuje więc raczej na możliwość wprowadzenia klas opisujących źródła przychodów i źródła ulg. Zwrot **„informacje o podatniku"** wskazuje na możliwość wprowadzenia klasy opisującej podatnika.

Czasowniki *ułatwia, pozwala* nie opisują bezpośrednio czynności wykonywanych przez system. Zwrot *„ułatwia* **przygotowywanie formularzy zeznań podatkowych"** informuje o tym, że system pozwala przygotowywać formularze.

Powyższy opis zawiera szereg rzeczowników odczasownikowych. Np. **przygotowywanie** — *przygotowywać*, **przechowywanie** — *przechowywać*, **tworzenie** — *tworzyć*.

Termin **wzorzec zeznania** odnosi się nie do klasy lecz do wydruku będącego wynikiem pewnej funkcji.

W rezultacie powyższej analizy w systemie tym można wskazać przedstawione poniżej potencjalne klasy:

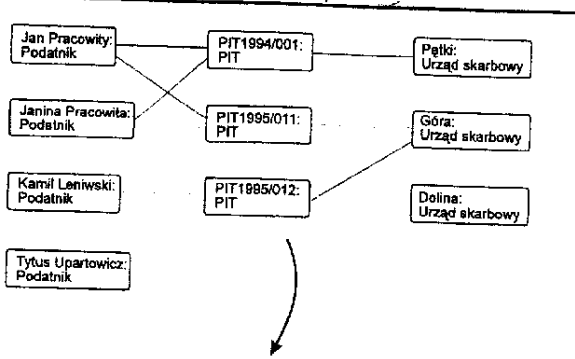
System podatkowy — potencjalne klasy

Program
PIT
Przychody z danej klasy — klasa opisująca przychody osiągnięte w ramach pewnej klasy źródeł
Przychody z konkretnego źródła
Roczne przychody
Ulgi podatkowe
Podatnik
Urząd skarbowy
Lista podatników
Lista urzędów skarbowych

Terminy: **kwota przychodu, kwota kosztów, kwota zapłaconych zaliczek, kwota dochodu, należny podatek, kwota do zapłaty, kwota do zwrotu** i **adres Urzędu Skarbowego** opisują raczej pola innych klas. Terminy: **pozarolnicza działalność gospodarcza, wolny zawód**, ..., mogą wskazywać na klasy będą specjalizacjami klasy Przychody z danej klasy. Klasy te jednak prawdopodobnie nie różniłyby się niczym między sobą, tj. miałyby dokładnie te same pola i metody. Ich wprowadzenie byłoby więc zbytnim rozbudowywaniem hierarchii generalizacji-specjalizacji. Lepszym rozwiązaniem będzie wprowadzenia pola Nazwa źródła w klasie Przychody z danej klasy.

Obiekty klas Podatnik oraz PIT mogą być ze sobą powiązane. Związki te wyrażają fakt opisywania przez dany formularz PIT dochodów pewnego podatnika(ów). Powiązane mogą być także obiekty klas PIT i Urząd skarbowy. Związki te wyrażają fakt wysyłania danego formularza PIT do pewnego Urzędu Skarbowego. Rysunek 5.47 przedstawia przykłady kilku konkretnych związków pomiędzy obiektami zaobserwowanych w firmie podatkowej. Z faktów tych wynika, że obiekt klasy PIT związany jest zawsze z jednym obiektem klasy Urząd skarbowy i z jednym lub dwoma obiektami klasy Podatnik. Obiekt klasy Podatnik związany jest z żadnym lub wieloma obiektami klasy PIT. Podobnie obiekt klasy Urząd skarbowy związany jest z żadnym lub wieloma obiektami klasy PIT.

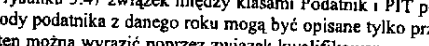
Szczególnie uważnie należy rozważyć obligatoryjność związków obiektów klasy PIT z obiektami klas Podatnik i Urząd skarbowy. Formularz PIT jest faktycznie przygotowywany zawsze dla jednego lub dwóch podatników, jest też zawsze wysyłany do pewnego Urzędu Skarbowego. Wątpliwość może budzić moment tworzenia obiektu klasy PIT.



Rysunek 5.47. Przykłady związków obiektów w systemie podatkowym

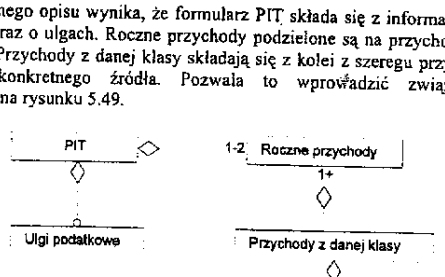
Obligatoryjność powyższych związków oznacza, że w momencie tworzenia nowego obiektu tej klasy, musi on zostać powiązany z obiektami (być może nowo utworzonymi) klas Podatnik i Urząd skarbowy. Rezygnacja z takiego powiązania musi być równoznaczna z zaniechaniem tworzenia nowego obiektu klasy PIT.

Wprowadzony na rysunku 5.47 związek między klasami Podatnik i PIT pomija pewien istotny fakt. Dochody podatnika z danego roku mogą być opisane tylko przez jeden formularz PIT. Fakt ten można wyrazić poprzez związek kwalifikowany przedstawiony na rysunku 5.48. Pozwoliło to ograniczyć krotność tego związku.



Rysunek 5.48. Kwalifikowany związek klas

Z przedstawionego opisu wynika, że formularz PIT składa się z informacji o rocznych przychodach oraz o ulgach. Roczne przychody podzielone są na przychody z poszczególnych klas. Przychody z danej klasy składają się z kolei z szeregu przychodów osiągniętych z konkretnego źródła. Pozwala to wprowadzić związki agregacji przedstawione na rysunku 5.49.



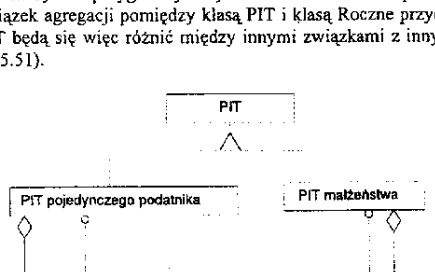
Rysunek 5.49. Związki agregacji w systemie podatkowym

Klasy Lista podatników i Lista urzędów skarbowych są typowymi przykładami zbiorów obiektów. Ponieważ w systemie pojawi się wiele obiektów klasy PIT, niezbędne jest także wprowadzenie klasy Lista PITów. Prowadzi to do związków przedstawionych na rysunku 5.50.

Rysunek 5.50. *Związki agregacji w systemie podatkowym*

Poszukując związków generalizacji-specjalizacji należałoby szczególną uwagę zwrócić na klasę PIT. Przedstawiony powyżej opis słowny mówi o dwóch rodzajach klasy formularzy PIT: formularzy opisujących dochody pojedynczego podatnika oraz formularzy opisujących dochody małżeństwa. Powstaje pytanie, czy można je uznać za różne klasy? Odpowiedź twierdząca wynika chociażby z analizy związku klasy PIT i klasy Podatnik (porównaj rysunki 5.47 i 5.48). Z wprowadzonego powyżej związku wynika, że obiekt klasy PIT jest związany z jednym lub dwoma obiektami Podatnik. Związek z jednym obiektem klasy Podatnik dotyczy jednak wyłącznie obiektów klasy PIT przygotowywanych dla pojedynczego podatnika. Związek z dwoma obiektami klasy Podatnik dotyczy wyłącznie obiektów klasy PIT przygotowywanych dla małżeństwa. W podobny można przeanalizować związek agregacji pomiędzy klasą PIT i klasą Roczne przychody. Specjalizacje klasy PIT będą się więc różnić między innymi związkami z innymi klasami (porównaj rysunek 5.51).

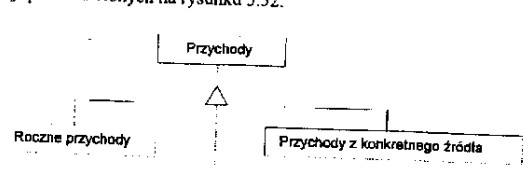
Rysunek 5.51. *Związki generalizacji-specjalizacji oraz zmodyfikowane związki klas w systemie podatkowym*



Gdyby specjalizacji klasy PIT nie wykryto przed próbą określenia pól tej klasy, okazałoby się, że pewne pola tej klasy, np. Adres męża i Adres żony nie zawsze miałyby sens. Świadczyłoby to o konieczności wprowadzenia specjalizacji tej klasy.

Klasy Roczne przychody, Przychody z danej klasy i Przychody z konkretnego źródła mają w nazwie ten sam człon. Wskazuje to na możliwość ich uogólnienia. Klasy te posiadają także te same pola: kwota przychodu, kwota kosztów, kwota zapłaconych zaliczek i kwota dochodu. Prowadzi to do wprowadzenia kolejnych związków generalizacji-specjalizacji przedstawionych na rysunku 5.52.

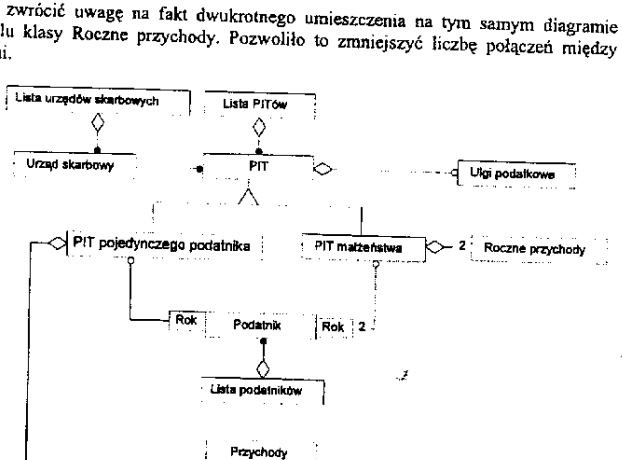
Rysunek 5.52. *Związki generalizacji-specjalizacji w systemie podatkowym*



Pełen diagram klas (bez pól i metod) systemu podatkowego przedstawiony jest na rysunku 5.53. Zaprezentowane związki klas nie są z pewnością jedynymi jakie można wyróżnić w tym systemie. Można np. wprowadzić związek klas Roczne przychody oraz Podatnik, wyrażający fakt opisywania przychodów pewnego podatnika. Byłby to jednak związek nadmiarowy w stosunku do istniejących już związków.

Warto zwrócić uwagę na fakt dwukrotnego umieszczenia na tym samym diagramie symbolu klasy Roczne przychody. Pozwoliło to zmniejszyć liczbę połączeń między klasami.

Rysunek 5.53. *Diagram klas systemu podatkowego*



Kolejnym krokiem analizy systemu podatkowego jest identyfikacja i definiowanie pól. Pola klas systemu podatkowego wraz z krótkim opisem wymienione są poniżej.

System podatkowy — pola klas

Urząd skarbowy
- Nazwa — ciąg znaków
- Adres — dana złożona

Podatnik
- Imię — ciąg znaków
- Nazwisko — ciąg znaków
- Imię ojca — ciąg znaków
- Imię matki — ciąg znaków
- Adres — dana złożona
- Data urodzenia — data
- Miejsce urodzenia — ciąg znaków
- PESEL — 11 cyfr

PIT
- Rok — liczba całkowita
- Podstawa opodatkowania — kwota pieniężna
- Należny podatek — kwota pieniężna
- Kwota do zwrotu — kwota pieniężna

PIT pojedynczego podatnika
- Adres — dana złożona

PIT małżeństwa
- Adres męża — dana złożona
- Adres żony — dana złożona

Przychody
- Kwota przychodu — kwota pieniężna
- Kwota kosztów — kwota pieniężna
- Kwota zaliczek — kwota pieniężna
- Kwota dochodu — kwota pieniężna

Przychody z danej klasy
- Nazwa źródła — ciąg znaków

Przychody z konkretnego źródła
- Opis — ciąg znaków
- Dokument — dana złożona

Ulgi podatkowe
- Kwota ulg — kwota pieniężna

Wprowadzenie pól Adres w klasach Podatnik, PIT pojedynczego podatnika i PIT małżeństwa może na pierwszy rzut oka wydawać się nadmiarowe. Podatnik może jednak zmieniać miejsce zamieszkania, pole Adres w klasie Podatnik służy więc do przechowywania aktualnego adresu podatnika. Pola Adres w klasie PIT pojedynczego podatnika oraz PIT małżeństwa służą do przechowywania adresów z dnia 31 grudnia danego roku podatkowego.

W klasach Lista urzędów skarbowych, Lista PITów, Lista podatników i Roczne przychody nie wprowadzono na etapie analizy żadnych pól. Klasa Roczne przychody dziedziczy jednak pola z klasy Przychody.

Pola klasy Przychody muszą spełniać podane poniżej ograniczenia:

Kwota przychodu, Kwota kosztów, Kwota zaliczek ≥ 0

Kwota przychodu > Kwota zaliczek

Kwota dochodu = Kwota przychodu – Kwota kosztów

Dodatkowe ograniczenia dotyczą pól klas Roczne przychody, Przychody z danej klasy oraz Przychody z konkretnego źródła:

Roczne przychody . Kwota przychodu = sumie pól Przychody z danej klasy . Kwota przychodu po wszystkich składowych obiektach

Przychody z danej klasy . Kwota przychodu = sumie pól Przychody z konkretnego źródła . Kwota przychodu po wszystkich składowych obiektach

W celu zilustrowania poszukiwania metod i komunikatów metodą analizy scenariuszy zostanie opracowany scenariusz przepływu komunikatów dla funkcji Edycja informacji o przychodach (porównaj sekcję 4.2.1 omawiającą hierarchię wymagań funkcjonalnych). Analiza tego scenariusza zostanie rozpoczęta od opracowania słownego opisu sposobu w jaki system realizuje tę funkcję:

Użytkownik wydaje polecenie edycji rocznych przychodów
- Wyświetlana jest lista przychodów z poszczególnych klas
- Użytkownik wydaje polecenie edycji przychodów z danej klasy
  - Zapamiętywane są aktualne kwoty opisujące przychody z wybranej klasy
  - Wyświetlana jest lista przychodów z konkretnych źródeł
  - Użytkownik wydaje polecenie dodania nowego przychodu z konkretnego źródła
    - Tworzony i edytowany jest opis nowego przychodu z konkretnego źródła
    - Uaktualniana jest lista przychodów z konkretnych źródeł
  - Użytkownik wydaje polecenie edycji przychodu z konkretnego źródła
    - Zapamiętywane są aktualne kwoty opisujące przychody z wybranego źródła
    - Edytowany jest opis przychodów z wybranego źródła
    - Na podstawie zapamiętanych kwot oraz kwot wyedytowanych przez użytkownika modyfikowane są kwoty opisujące przychody z danej klasy
    - Uaktualniana jest lista przychodów z konkretnych źródeł

Użytkownik wydaje polecenia usunięcia przychodu z konkretnego źródła

Kwoty opisujące przychody z danej klasy pomniejszane są o kwoty opisujące przychody z wybranego źródła

Usuwany jest opis przychodów z wybranego źródła

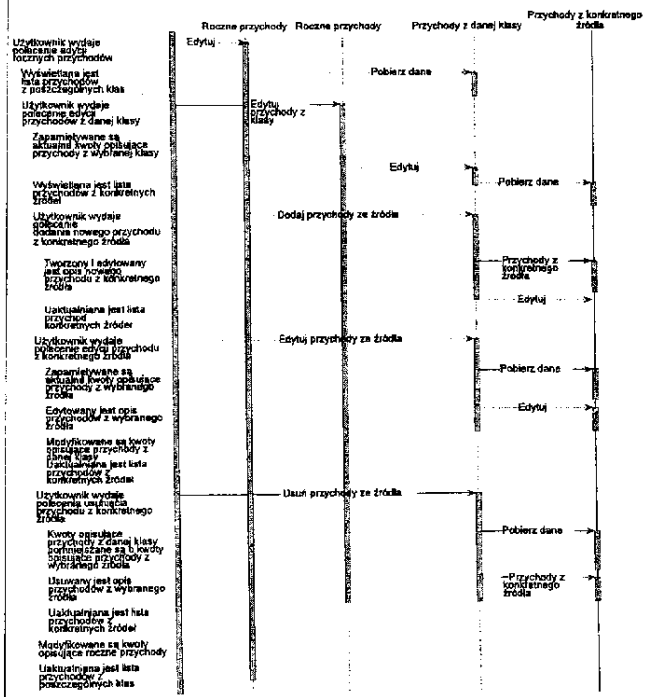
Uaktualniana jest lista przychodów z konkretnych źródeł

Po zakończeniu edycji przychodów z danej klasy na podstawie zapamiętanych kwot oraz kwot wyedytowanych przez użytkownika modyfikowane są kwoty opisujące roczne przychody

Uaktualniana jest lista przychodów z poszczególnych klas

Scenariusz przepływu komunikatów odpowiadający temu opisowi przedstawiony jest na rysunku 5.54. Scenariusz ten zakłada, że metody Edytuj klas Przychody z danej klasy oraz Przychody z konkretnego źródła zwracają jako dane wyjściowe wyedytowane przez użytkownika kwoty przychodu, dochodu, zaliczek i kosztów.

Rysunek 5.54. *Scenariusz przepływu komunikatów realizujący funkcję „Edycja informacji o przychodach"*

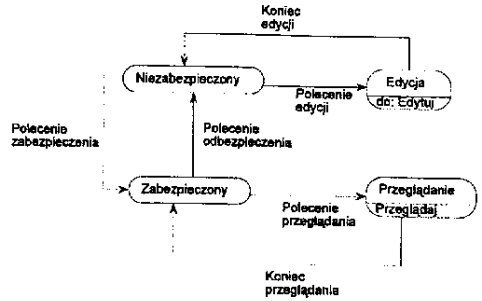


Wysłanie komunikatu wywołującego konstruktor pozwala zobrazować fakt tworzenia nowego obiektu danej klasy.

Klasą, której obiekty niewątpliwie zmieniają swój stan, jest PIT. Obiekt tej klasy może zostać zabezpieczony przed dalszymi zmianami. Czynność ta powinna być wykonana po wysłaniu formularza PIT do Urzędu Skarbowego.

W przypadku zabezpieczonego obiektu tej klasy możliwe jest jedynie przeglądanie jego opisu. Model przejść stanów klasy PIT przedstawiony jest na rysunku 5.55. Model ten dopuszcza odbezpieczenie obiektu tej klasy.

Rysunek 5.55. *Model przejść stanów klasy PIT*



Stany Edycja i Przeglądanie odpowiadają okresom, w których wykonywane są pewne operacje. Nie dotyczy to stanów Niezabezpieczony i Zabezpieczony. Niezbędne jest więc wprowadzenie dodatkowego pola klasy PIT, na przykład Stan, które będzie informowało o tym, w którym z tych stanów znajduje się dany obiekt.

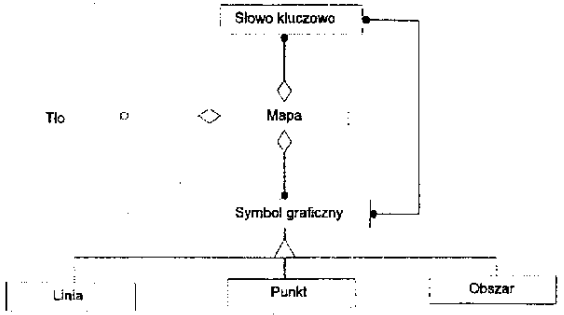
5.4.3.2. System informacji geograficznej

W systemie tym można wyróżnić następujące klasy:

System informacji geograficznej — potencjalne klasy

Mapa
Tło
Obiekt graficzny
Punkt
Linia
Obszar
Słowo kluczowe

Rysunek 5.56 prezentuje związki klas tego systemu.



Rysunek 5.56. Diagram klas systemu informacji geograficznej

Wyraża on następujące fakty:

- Mapa składa się opcjonalnie z jednego tła, z wielu słów kluczowych oraz z wielu symboli graficznych.
- Rodzajami symboli graficznych są punkty, linie lub obszary.
- Symbol graficzny może być opcjonalnie opisywany przez wiele słów kluczowych.
- Słowo kluczowe może opcjonalnie opisywać wiele symboli graficznych.

Poniżej wymienione są wraz z krótkim opisem pola klas systemu informacji geograficznej.

System informacji geograficznej — pola klas

Mapa
- Nazwa — ciąg znaków
- Wyświetlany obszar — prostokąt (dana złożona)

Tło
- Nazwa pliku — identyfikator pliku

Słowo kluczowe
- Nazwa — ciąg znaków
- Opis — ciąg znaków
- Stan — wybrane lub niewybrane

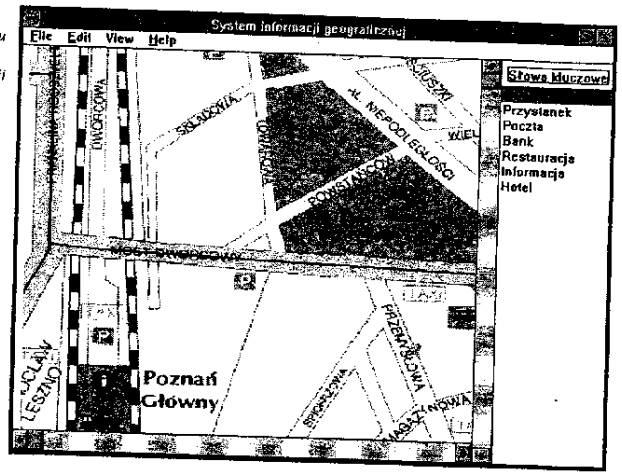
Symbol graficzny
- Nazwa — ciąg znaków
- Opis — polecenie systemu operacyjnego wywołujące zewnętrzny program wyświetlający opis symbolu

Punkt
- Współrzędna pozioma — liczba całkowita
- Współrzędna pionowa — liczba całkowita
- Znak — nazwa pliku zawierającego symbol punktu w formacie WMF (Windows Metafile)

Linia
- Lista punktów — lista współrzędnych punktów tworzących linię (dana złożona)
- Wzór — identyfikator wzoru linii
- Kolor — identyfikator koloru linii

Obszar
- Lista punktów — lista współrzędnych punktów tworzących linię ograniczającą obszar (dana złożona)
- Wypełnienie — identyfikator wypełnienia obszaru
- Kolor — identyfikator koloru obszaru

W dalszym ciągu zostanie przeanalizowany scenariusz przepływu komunikatów realizujący funkcję Przeglądanie SIG (porównaj sekcję 4.2.1 omawiającą hierarchię wymagań funkcjonalnych). Zostaną przy tym poczynione pewne założenia dotyczące interfejsu użytkownika (patrz rysunek 5.57).



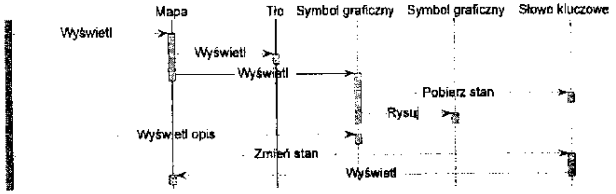
Rysunek 5.57. Prototyp interfejsu użytkownika systemu informacji geograficznej

Mapa wyświetlana jest w pewnym oknie. Użytkownik może przewijać i wyświetlać w różnych powiększeniach zawartość tego okna. Obok znajduje się lista słów kluczowych. W sposób wizualny wyróżnione są aktualnie wybrane i niewybrane słowa kluczowe. Użytkownik zmienia stan słowa kluczowego klikając myszą na tym słowie. Użytkownik może wyświetlić opis danego obiektu graficznego klikając na tym obiekcie.

Z powyższego opisu wynika, że w trakcie realizacji tej funkcji użytkownik może wydać trzy polecenia:

- Wyświetl mapę
- Wyświetl opis symbolu graficznego
- Zmień stan słowa kluczowego

Na rysunku 5.58 przedstawiony jest diagram interakcji opisujący sposób realizacji tej funkcji. Przedstawiony na tym rysunku scenariusz zakłada, że metody Wyświetl klas Mapa, Tło i Symbol graficzny otrzymują jako daną wejściową obszar mapy, który należy wyświetlić. W przypadku metody Wyświetl klasy Mapa jest to dana opcjonalna. Jeżeli nie jest podana, wyświetlany jest obszar zapamiętany w polu Wyświetlany obszar.



Rysunek 5.58. Scenariusz przepływu komunikatów realizujący funkcję „Przeglądanie SIG"

Zmiana stanu obiektu klasy Słowo kluczowe może spowodować, konieczność wyświetlenia lub ukrycia pewnych symboli graficznych. Konieczne jest więc przerysowanie mapy co zapewnia wysłanie komunikatu Wyświetl przez obiekt klasy Słowo kluczowe do obiektu klasy Mapa. Scenariusz ten zakłada, że obiekt klasy Symbol graficzny wywołuje swoją własną metodę Rysuj. Metoda Rysuj klasy Symbol graficzny jest metodą wirtualną redefiniowaną w jej specjalizacjach. Jest to metoda odpowiedzialna za faktyczne przerysowanie symbolu jeżeli mieści się w ramach wyświetlanego obszaru. Metoda Wyświetl tej klasy jest odpowiedzialna za sprawdzenie czy dany obiekt powinien być wyświetlony.

Poniżej podano przykłady specyfikacji metod występujących na powyższym diagramie.

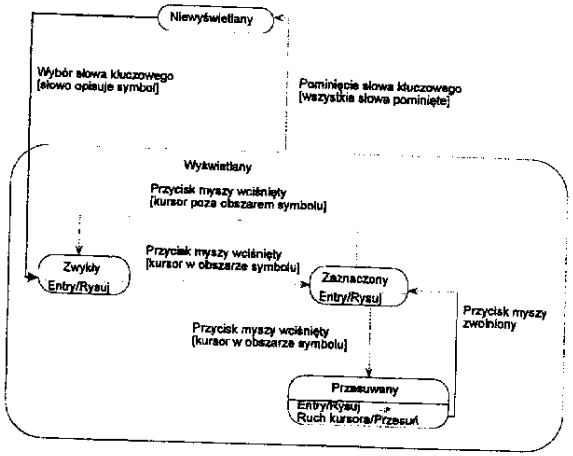
Specyfikacja metody Wyświetl klasy Mapa

Dane wejściowe
Wyświetlany obszar — prostokąt (dana złożona)
Algorytm
jeżeli istnieje składowy obiekt klasy Tło, **to**
wyślij do tego obiektu komunikat Wyświetl (Wyświetlany obszar)
dla każdego składowego obiektu klasy Symbol graficzny **powtarzaj**
wyślij do tego obiektu komunikat Wyświetl (Wyświetlany obszar)

Specyfikacja metody Wyświetl klasy Symbol graficzny

Dane wejściowe
Wyświetlany obszar — prostokąt (dana złożona)
Algorytm
dla każdego powiązanego obiektu klasy Słowo kluczowe **powtarzaj**
Pobierz stan słowa kluczowego
jeżeli słowo kluczowe jest wybrane, **to**
wywołaj metodę Rysuj (Wyświetlany obszar)
dopóki nie znaleziono wybranego słowa kluczowego

Rysunek 5.59 prezentuje model przejść stanów klasy Symbol graficzny w trakcie edycji mapy. Obiekt tej klasy może być wyświetlany lub niewyświetlany w zależności od stanu obiektów klasy Słowo kluczowe, związanych z tym obiektem.



Rysunek 5.59. Model przejść stanów klasy Symbol graficzny

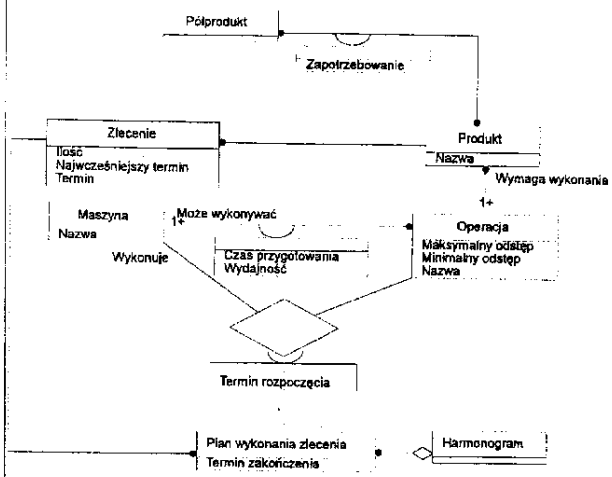
Są to więc przykłady stanów opisywanych przez stan związków danego obiektu oraz zawartość pól powiązanych z nim obiektów. Stan Wyświetlany jest uogólnieniem trzech innych stanów. Rozróżnienie stanów Zwykły, Zaznaczony i Przesuwany wymaga wprowadzenia dodatkowego pola.

5.4.3.3. System harmonogramowania zleceń

Rysunek 5.60 zawiera diagram klas systemu harmonogramowania zleceń. Przedstawione są także pola poszczególnych klas. Na diagramie tym pominięto klasy będące prosty-

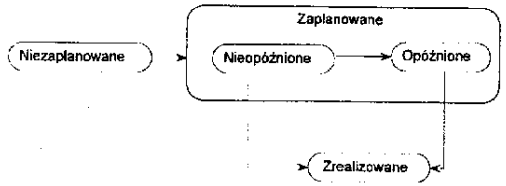
mi zbiorami obiektów, na przykład Lista zleceń, Lista produktów, itd. Diagram ten zawiera kilka związków, w tym związek ternarny, opisanych polami.

Rysunek 5.60. Diagram klas systemu harmonogramowania zleceń



Rysunek 5.61 przedstawia model przejść stanów klasy Zlecenie.

Rysunek 5.61. Model przejść stanów klasy Zlecenie



5.5. Notacje strukturalne i ich interpretacja

Cechą charakterystyczną metod strukturalnych jest fakt posługiwania się w fazach analizy i projektowania odmiennymi notacjami i pojęciami. Nie oznacza to jednak, że notacje strukturalne stosowane w fazie analizy są bardziej odległe od poziomu kodu niż notacje obiektowe. Przeciwnie, pewne notacje, jak np. diagramy związków encji są bardzo silnie ukierunkowane na modelowanie struktury relacyjnych baz danych.

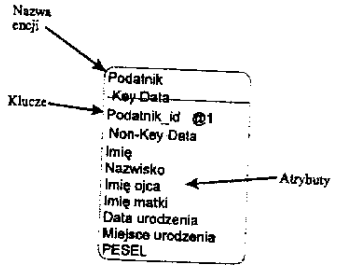
5.5.1. Diagramy związków encji

Diagramy związków encji (ang. *entity relation diagrams*) służą do prezentowania danych przechowywanych w systemie oraz związków pomiędzy nimi. Model związków encji nazywa się też modelem danych lub modelem informacyjnym. Diagramy te są podobne do diagramów klas omówionych w rozdziale 5.3.1.

Encja (ang. *entity*)

Zbiór obiektów, czyli wystąpień encji, posiadających tożsamość i stan. Encja składa się z atrybutów (ang. *attributes*). Atrybuty opisują stan obiektów encji. Atrybut nazywany jest kluczem jeżeli w sposób jednoznaczny identyfikuje poszczególne wystąpienia encji, tj. przyjmuje unikalne wartości w zbiorze wystąpień encji. Symbol encji przedstawiony jest na rysunku 5.62. Klucze i atrybuty mogą zostać pominięte.

Rysunek 5.62. Symbol encji



Pojęcie encji jest bardzo zbliżone do pojęcia klasy. Istnieją jednak pomiędzy nimi pewne istotne różnice wymienione w tabeli 5.2.

Tabela 5.2. Encja i klasa — porównanie

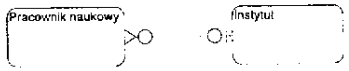
Encja	*Klasa*
Pasywna	Aktywna
Jest wzorcem oraz zbiorem obiektów	Jest wzorcem obiektów
Trwała	Trwała lub ulotna
Musi zawierać atrybuty	Dopuszczalne są klasy nie posiadające atrybutów
Atrybuty elementarne	Pola elementarne i złożone
Musi posiadać unikalny identyfikator — klucz	Obiekty klasy są rozróżnialne niezależnie od wartości pól

Encja jest składową pasywną gdyż opisuje wyłącznie dane przechowywane w systemie. Metody mogą być czynnikiem rozróżniającym klasy o takich samych polach i związkach. Ponieważ encja jest zbiorem swoich wystąpień, do modelu związków encji nie wprowadza się encji będących prostymi zbiorami obiektów, jak miało to miejsce w przypadku modelu obiektowego. Encje wykorzystuje się praktycznie wyłącznie do modelowania danych trwałych. Dane przechowywane w polach klas mogą być zarówno trwałe, jak i ulotne. Wprowadzanie encji nie posiadających atrybutów nie miało by oczywiście sensu. W przypadku klas może to się zdarzyć, szczególnie w przypadku klas będących interfejsami dla systemów zewnętrznych lub urządzeń sprzętowych. Założenie o elementarności atrybutów encji wiąże się z utożsamianiem encji i relacji w relacyjnej bazie danych. Wiele narzędzi CASE pozwala jednak obecnie na posługiwanie się atrybutami złożonymi. W momencie tworzenia schematu bazy danych system w sposób automatyczny eliminuje atrybuty złożone, np. rozbijając je na kilka atrybutów elementarnych.

Związek encji (ang. *entity relation*)

Związek ten jest odpowiednikiem związku klas wykorzystywanego w modelu obiektowym. Rysunek 5.63 przedstawia związek opisujący fakt, że pracownicy naukowi uczelni pracują w instytutach, z których składa się ta uczelnia (porównaj rysunek 5.8 prezentujący ten sam związek w modelu obiektowym).

Rysunek 5.63.
Przykład związku encji



Podobnie jak w przypadku związków encji symbol związku zawiera dodatkowe oznaczenia informujące o jego krotności. Oznaczenia te są zebrane w tabeli 5.3.

Tabela 5.3.
Oznaczenia krotności związków encji

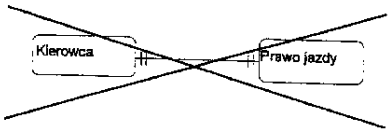
Krotność	*Oznaczenie*
Dokładnie jeden	
Zero lub wiele	
Jeden lub wiele	
Zero lub jeden	
Więcej niż jeden	
Nieoznaczona	

Podobnie jak związki klas związki encji można opisywać podając:

- nazwę związku, np. zatrudnianie, wykładanie,
- rolę (role) odgrywane w związku przez wystąpienia danej encji,
- czasownik opisujący czynność (operację, zadanie) wykonywaną w ramach związku przez wystąpienia (obiekty) danej encji.

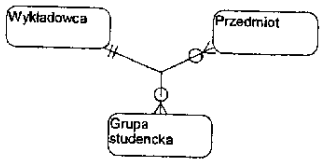
W przypadku modelu związków encji z reguły nie wprowadza się związków o obustronnej krotności dokładnie jeden (porównaj rysunek 5.64). Dotyczy to również związków o jednostronnej, stałej krotności większej od jeden. Jeżeli np. obiekt pewnej encji związany jest zawsze z trzema obiektami innej encji, które z kolei związane są zawsze tylko z tym jednym obiektem, to można wprowadzić pojedynczą encję zawierającą atrybuty wszystkich powiązanych obiektów. Zamiana każdej z pozostających w tym związku encji na odrębną relację w bazie danych, prowadziłaby bowiem do zmniejszenia efektywności systemu.

Rysunek 5.64.
W modelu związków encji nie wprowadza się związków o obustronnej krotności dokładnie jeden



Model związków encji dopuszcza także związki ternarne (porównaj rysunek 5.65).

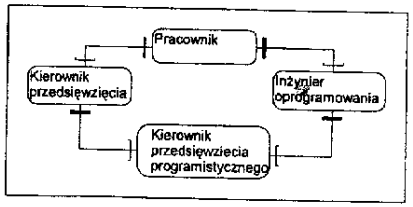
Rysunek 5.65.
Symbol związku ternarnego



Związek generalizacji-specjalizacji (ang. *generalization-specialization relationship, is-a relationship*)

Związek ten jest odpowiednikiem związku o tej samej nazwie w modelu obiektowym. Symbol tego związku przedstawiony jest na rysunku 5.66.

Rysunek 5.66.
Przykłady związków generalizacji-specjalizacji



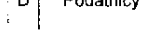
5.5.2. Diagramy przepływów danych

Diagramy przepływów danych (ang. *data flow diagrams*) służą do prezentowania sposobu w jaki dane przepływają oraz są przetwarzane w systemie. Opisują również procesy przetwarzające dane.

Zbiornik danych (magazyn danych, ang. *data store*)

Jest to składowa systemu przechowująca dane. Zbiornik danych może przechowywać dane trwałe lub ulotne. Symbol zbiornika danych przedstawiony jest na rysunku 5.67.

Rysunek 5.67. Symbol zbiornika danych



Proces (ang. *process*)

Operacja (czynność) wykonywana przez system. Symbol procesu przedstawiony jest na rysunku 5.68.

Rysunek 5.68. Symbol procesu

System zewnętrzny (ang. *external system*)

System zewnętrzny, z którym system wymienia dane. Symbol systemu zewnętrznego przedstawiony jest na rysunku 5.69.

Rysunek 5.69. Symbol systemu zewnętrznego

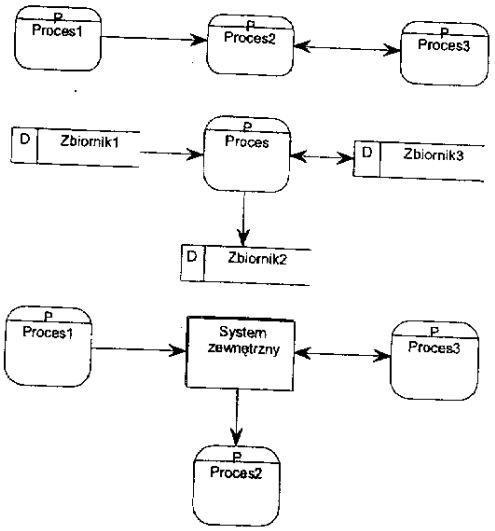
Przepływ danych (ang. *data flow*)

Opisuje dane przesyłane pomiędzy procesami, systemami zewnętrznymi i zbiornikami danych. Przepływ danych może być jedno- lub dwukierunkowy. Przepływ pomiędzy procesami oznacza fakt przekazywania danych pomiędzy tymi procesami. Nie jest przy tym określone w jaki sposób następuje przekazywanie danych. Przepływ pomiędzy procesem i zbiornikiem danych oznacza fakt odczytywania, zapisywania lub modyfikowania przez proces danych zawartych w zbiorniku. Przepływ pomiędzy procesem i systemem zewnętrznym opisuje fakt wymiany danych z tym systemem. Symbol przepływu danych oraz dopuszczalne przepływy przedstawione są na rysunku 5.70.

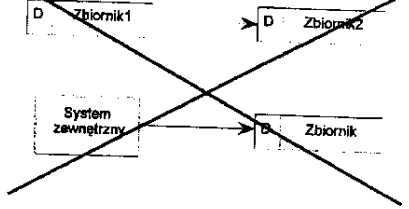
Niedopuszczalne są przepływy pomiędzy dwoma zbiornikami oraz systemem zewnętrznym i zbiornikiem danych. Niedopuszczalne przepływy danych przedstawione są na rysunku 5.71.

Proces umieszczony na pewnym diagramie może być w sposób bardziej szczegółowy opisany przez diagram przepływów danych niższego poziomu. Diagramy przepływów danych tworzą więc hierarchię. Na poziomie 0 — to jest najwyższym — znajduje się

Rysunek 5.70. Dopuszczalne przepływy danych

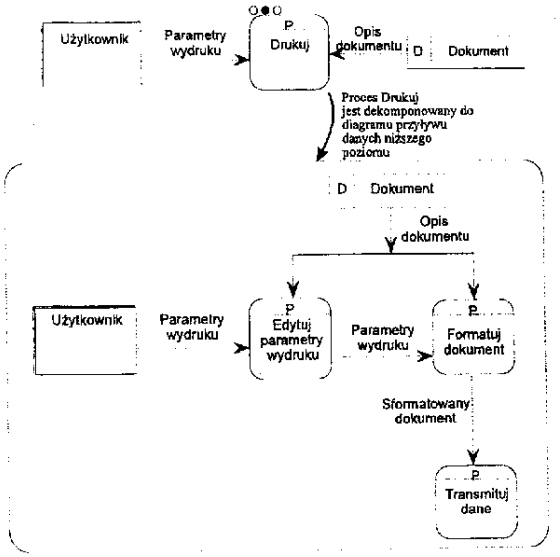


Rysunek 5.71. Niedopuszczalne przepływy danych



tak zwany diagram kontekstowy. Na diagramie tym cały system opisany jest przez pojedynczy proces. Zawiera on również informacje o przepływach danych pomiędzy modelowanym systemem a systemami zewnętrznymi. Na poziomie 1 znajduje się diagram obejmujący główne procesy systemu, które z kolei opisywane są przez diagramy kolejnych, niższych poziomów. Każdy diagram, z wyjątkiem diagramu kontekstowego ma więc jeden proces będący jego rodzicem. Każdy proces może mieć jeden diagram będący jego potomkiem.

Przykład hierarchii diagramów przepływów danych jest przedstawiony na rysunku 5.72. Trzy kropki nad symbolem procesu oznaczają, że posiada on diagram potomny.



Rysunek 5.72. Przykład hierarchii diagramów przepływów danych

5.5.3. Diagramy przejść stanów

Diagramy przejść stanów, omówione już w rozdziale 5.3.3, dla przypadku obiektowego, stosowane są także w metodach strukturalnych. Stosowane są one do modelowania dynamicznego zachowania całego systemu, grup lub pojedynczych encji oraz procesów.

5.5.4. Specyfikacja modelu strukturalnego

Podobnie jak w przypadku modelu obiektowego typowymi składowymi specyfikacji, stosowanymi do opisu wszystkich elementów są: nazwa oraz ogólny opis.

Proces specyfikuje się przede wszystkim opisując algorytm jego działania. Stosuje się przy tym podobne techniki jak w przypadku modelu obiektowego (patrz sekcja 5.3.4 omawiająca specyfikację modelu obiektowego). W przypadku metod strukturalnych większy nacisk kładzie się na specyfikację algorytmu już na poziomie analizy. Diagramy przepływów danych nie zawierają bowiem informacji o kolejności oraz sposobie wywoływania poszczególnych procesów. Dodatkowo dla procesów specyfikuje się:

- warunki wstępne, to jest warunki jakie muszą spełniać dane zawarte w zbiornikach danych i encjach, z których korzysta proces oraz dane wejściowe, aby proces mógł rozpocząć się poprawnie,

- warunki końcowe, to jest warunki jakie muszą spełniać dane zawarte w zbiornikach danych i encjach, z których korzysta proces oraz dane wyjściowe po zakończeniu poprawnie rozpoczętego oraz poprawnie wykonanego procesu,
- wyjątki, to jest sytuacje wyjątkowe, które mogą zajść w trakcie realizacji procesu (np. błąd odczytu pliku),
- złożoność czasowa,
- złożoność pamięciowa.

Dane, to jest encje, zbiorniki danych oraz przepływy danych specyfikuje się w dokładnie ten sam sposób jak w przypadku modelu obiektowego (porównaj sekcję 5.3.4 omawiającą specyfikację modelu obiektowego).

5.6. Proces tworzenia modelu strukturalnego

Proces tworzenia modelu strukturalnego polega na wykonaniu dwóch głównych zadań:

- modelowania danych,
- modelowania procesów i przepływów danych.

Pierwsze z tych zadań odpowiada budowie statycznego modelu klas (porównaj sekcję 5.4 omawiającą proces tworzenia modelu obiektowego) i obejmują:

- identyfikację encji,
- identyfikację związków encji,
- identyfikację i definiowanie atrybutów.

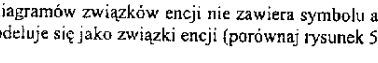
Tradycyjne podejście zakłada, że dwie główne czynności, tj. modelowanie danych oraz modelowanie procesów i przepływów danych powinny być wykonywane zupełnie niezależnie, najlepiej przez różne zespoły analityków. Podejście to jest ostro krytykowane przez zwolenników metod obiektowych, którzy zwracają uwagę na ryzyko braku spójności pomiędzy danymi pojawiającymi się w obu modelach, to jest encjami i zbiornikami danych. Zwolennicy metod strukturalnych argumentują jednak, że analiza systemu z dwóch odmiennych punktów widzenia, pozwala na lepsze zrozumienie wszystkich aspektów systemu.

Obecnie coraz częściej zaleca się ścisłą integrację modelowania danych oraz modelowania procesów i przepływów danych. Zaleca się wykonywanie tych dwóch zadań przez te same grupy analityków oraz ścisłą synchronizację opisów encji i zbiorników danych. Zapewnienie takiej synchronizacji zdecydowanie ułatwiają strukturalne narzędzia CASE.

5.6.1. Modelowanie danych

Proces modelowania danych jest bardzo zbliżony do procesu budowy statycznego modelu klas omówionego w sekcji 5.4.1. Różnice na jakie należy zwrócić uwagę to (porównaj różnice pomiędzy encją a klasą omówione w sekcji 5.5.1):

- poszukując encji zwraca się uwagę na dane trwałe,
- nie wprowadza się encji nie posiadających atrybutów,
- nie wprowadza się encji będących prostymi zbiorami obiektów,
- tradycyjnie mniejszą uwagę przywiązuje się do modelowania związków generalizacji-specjalizacji,
- ponieważ notacja diagramów związków encji nie zawiera symbolu agregacji, tego typu związki modeluje się jako związki encji (porównaj rysunek 5.73).



Rysunek 5.73. Związek encji opisujący agregację

5.6.2. Modelowanie procesów i przepływów danych

Jedno z typowo stosowanych podejść do modelowania procesów i przepływów danych to podejście zstępujące wykonywane według następującego schematu:

- Skonstruuj diagram kontekstowy obejmujący systemy zewnętrzne, pojedynczy proces opisujący cały system oraz przepływy danych pomiędzy systemami zewnętrznymi oraz systemem.
- Zdekomponuj proces znajdujący się na powyższym diagramie do opisującego go diagramu przepływu danych.
- Powtarzaj dekompozycję dla każdego procesu aż do osiągnięcia poziomu procesów elementarnych.

Przepływy danych wpływające i wypływające z procesu, który ma diagram potomny powinny mieć swoje odpowiedniki na tym diagramie. Na diagramie potomnym przepływy takie powinny łączyć procesy z systemami zewnętrznymi i zbiornikami danych pojawiającymi się już na diagramie wyższego poziomu (porównaj rysunek 5.72). Możliwe jest także rysowanie ich jako przepływów połączonych tylko jednostronnie z pewnym procesem, wpływających i wypływających na zewnątrz diagramu.

Przepływ danych wpływający lub wypływający z procesu rodzica może na diagramie potomnym być zdekomponowany na kilka przepływów. Powinny one jednak nieść w sumie te same dane co przepływ na diagramie wyższego poziomu.

Jedna z technik dekompozycji polega na wykorzystaniu tekstu w języku naturalnym, zawierającego krótki opis dekomponowanego procesu. Czasowniki wyróżnione w tym opisie wskazują na potencjalne procesy, rzeczowniki na zbiorniki i przepływy danych. Oczywiście należy tu zwrócić uwagę na niejednoznaczność i elastyczność języka naturalnego (porównaj sekcję 5.4.1.1 omawiającą podobne podejście stosowane do identyfikacji klas). Opis procesu może być mniej lub bardziej precyzyjny. W zależności od tego diagram potomny będzie mniej lub bardziej szczegółowy. Jedną z zasad, która pozwala określić stopień szczegółowości diagramu potomnego jest wywodząca się z badań psychologicznych reguła 7±2 (Miller, 1956). Zgodnie z tą regułą liczba obiektów, nad którymi człowiek jest w stanie jednocześnie pracować waha się pomiędzy 5 a 9. Liczba procesów i zbiorników danych znajdujących się na pojedynczym diagramie powinna się więc również mieścić w tych granicach. Fakt, że poszczególne procesy mogą być przez różnych analityków dekomponowane z różnym poziomem szczegółowości może prowadzić do wyraźnie odmiennych, choć równoważnych sobie, hierarchii przepływów danych. Obserwacja ta jest jedną z podstaw krytyki metod strukturalnych przez zwolenników podejścia obiektowego twierdzących, że hierarchia diagramów przepływów danych nie odzwierciedla rzeczywistych zależności w dziedzinie problemu.

Inna technika dekompozycji ściśle wiąże procesy z wymaganiami funkcjonalnymi. Proces pojawiający się na diagramie kontekstowym odpowiada ogólnej funkcji systemu. Procesy pojawiające się na diagramie kolejnego poziomu odpowiadają funkcjom na jakie dekomponowana jest funkcja ogólna, itd. Oczywiście proces dekompozycji nie kończy się po osiągnięciu poziomu elementarnych wymagań funkcjonalnych. Odpowiadające im procesy mogą być nadal bardzo skomplikowane i wymagać dalszej dekompozycji.

Należy oczywiście określić jakie procesy uznaje się za elementarne i nie wymagające dalszej dekompozycji. Podstawowym kryterium jest ich prostota. Proces, którego algorytm jest na tyle prosty, że będzie mógł zostać bez trudu zaimplementowany można uznać za elementarny. Oczywiście nie jest to kryterium niezależne od cech środowiska implementacji oraz stopnia zaawansowania osób odpowiedzialnych za implementację.

Inne często stosowane podejście do modelowania procesów i przepływów danych jest opisane przez następujący schemat:

- Wybierz jedną z funkcji (niekoniecznie elementarną) realizowaną przez system.
- Określ komunikaty, które otrzymuje system podczas realizacji tej funkcji.
- Dla każdego komunikatu wprowadź proces będą reakcją na ten komunikat.
- Dodaj zbiorniki danych oraz przepływy danych pomiędzy procesami oraz zbiornikami danych.
- Połącz powiązane procesy i zbiorniki danych w procesy wyższego poziomu aż do osiągnięcia poziomu diagramu kontekstowego.
- Zdekomponuj procesy, których nie można uznać za elementarne.

Jest to więc technika mieszana, zarówno wstępująca, jak i zstępująca.

Na uwagę zasługuje kwestia stosunku encji oraz zbiorników danych. Jak już wspomniano na wstępie omawiania procesu budowy modelu strukturalnego, obecnie często zaleca się ścisłą synchronizację opisów encji i zbiorników danych. Niektórzy idą jeszcze dalej postulując wzajemną jednoznaczność tych dwóch składowych modelu. Każdej encji powinien więc odpowiadać jeden zbiornik danych. Każdemu zbiornikowi danych służącemu do przechowywania danych trwałych powinna odpowiadać jedna encja. Nie jest to jednak postulat powszechnie akceptowany. Pewnym kompromisem może być wprowadzanie zbiorników danych obejmujących zawsze jedną lub kilka pełnych encji. Procesy znajdujące się na diagramach wyższego poziomu będą często korzystały z danych zawartych w wielu encjach. Wprowadzenie zbiornika danych dla każdej z tych encji prowadziłoby do zbytniego rozbudowania tych diagramów. Wprowadza się więc zbiorniki danych obejmujące kilka encji. Na niższym poziomie w hierarchii diagramów procesy będą raczej korzystały z danych zawartych w pojedynczych encjach. Zbiorniki danych umieszczone na tych diagramach mogą więc odpowiadać pojedynczym encjom.

5.6.3. Modelowanie przejść stanów

Technika ta została już opisana w sekcji 5.4.2.2. Tę samą technikę stosuje się zarówno w budowie modelu strukturalnego, jaki i obiektowego. Wynikiem jej stosowania może być wykrycie dodatkowych procesów i danych.

5.6.4. Przykład

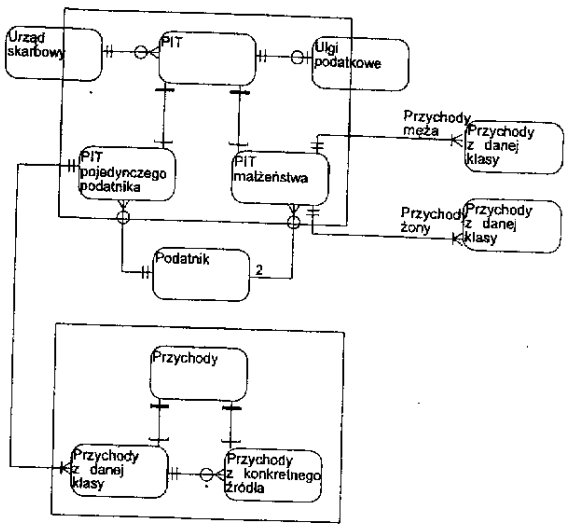
Rysunek 5.74 zawiera diagram związków encji systemu podatkowego. Diagram ten jest oczywiście podobny do diagramu klas dla tego samego systemu (patrz rysunek 5.53).

Na diagramie związków encji nie wprowadzono jednak encji będących prostymi zbiorami obiektów. Nie wprowadzono także encji Roczne przychody. Pomiędzy taką encją istniałby bowiem związek o obustronnej krotności dokładnie jeden z encją PIT pojedynczego podatnika oraz związek o stałej krotności dwa z encją PIT małżeństwa. Atrybuty encji Roczne przychody zostały więc włączone do encji PIT pojedynczego podatnika i PIT małżeństwa, przy czym w tej drugiej encji zostały one powielone dwukrotnie dla męża i żony. W rezultacie uzyskano następujące atrybuty encji:

System podatkowy — atrybuty encji

Urząd skarbowy
- Nazwa — ciąg znaków
- Adres — dana złożona

Podatnik
- Imię — ciąg znaków
- Nazwisko — ciąg znaków
- Imię ojca — ciąg znaków
- Imię matki — ciąg znaków



Rysunek 5.74. Diagram związków encji systemu podatkowego

- Adres — dana złożona
- Data urodzenia — data
- Miejsce urodzenia — ciąg znaków
- PESEL — 11 cyfr

PIT
- Rok — liczba całkowita
- Podstawa opodatkowania — kwota pieniężna
- Należny podatek — kwota pieniężna
- Kwota do zwrotu — kwota pieniężna
- Stan — {Zabezpieczony, Niezabezpieczony}

PIT pojedynczego podatnika
- Adres — dana złożona
- Kwota rocznego przychodu — kwota pieniężna
- Kwota rocznego kosztów — kwota pieniężna
- Kwota rocznych zaliczek — kwota pieniężna
- Kwota rocznego dochodu — kwota pieniężna

PIT małżeństwa
- Adres męża — dana złożona
- Adres żony — dana złożona
- Kwota rocznego przychodu męża — kwota pieniężna

Kwota rocznych kosztów męża — kwota pieniężna
Kwota rocznych zaliczek męża — kwota pieniężna
Kwota rocznego dochodu męża — kwota pieniężna
Kwota rocznego przychodu żony — kwota pieniężna
Kwota rocznych kosztów żony — kwota pieniężna
Kwota rocznych zaliczek żony — kwota pieniężna
Kwota rocznego dochodu żony — kwota pieniężna

Przychody
Kwota przychodu — kwota pieniężna
Kwota kosztów — kwota pieniężna
Kwota zaliczek — kwota pieniężna
Kwota dochodu — kwota pieniężna

Przychody z danej klasy
Nazwa klasy — ciąg znaków

Przychody z konkretnego źródła
Opis — ciąg znaków
Dokument — dana złożona

Ulgi podatkowe
Kwota ulg — kwota pieniężna

Rysunki 5.75 – 5.77 zawierają fragment hierarchii diagramów systemu podatkowego począwszy od diagramu kontekstowego. Poszczególne procesy odpowiadają funkcjom systemu (porównaj sekcję 4.2.1, omawiającą hierarchię wymagań funkcjonalnych). Zbiornik danych Zeznania podatkowe obejmuje encje PIT pojedynczego podatnika, PIT małżeństwa, Przychody z danej klasy, Przychody z konkretnego źródła, Ulgi podatkowe.

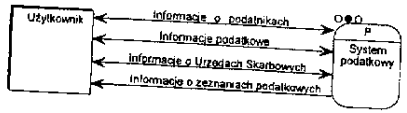
5.7. Podsumowanie

5.7.1. Kluczowe czynniki sukcesu

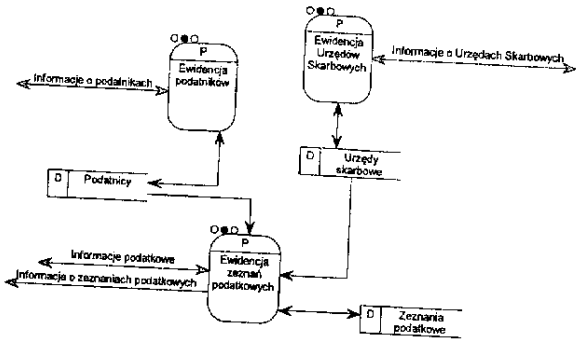
Najistotniejsze elementy wpływające na sukces fazy analizy to:

- *Zaangażowanie właściwych osób ze strony klienta.*
- *Zachowanie przyjętych standardów* dotyczących np. stosowanych notacji.
- *Sprawdzenie poprawności modelu.* Pewne błędne konstrukcje w poszczególnych notacjach zostały wymienione w poprzednich rozdziałach. Zagadnienia poprawności modelu i projektu zostaną szczegółowo omówione w sekcji 6.6.

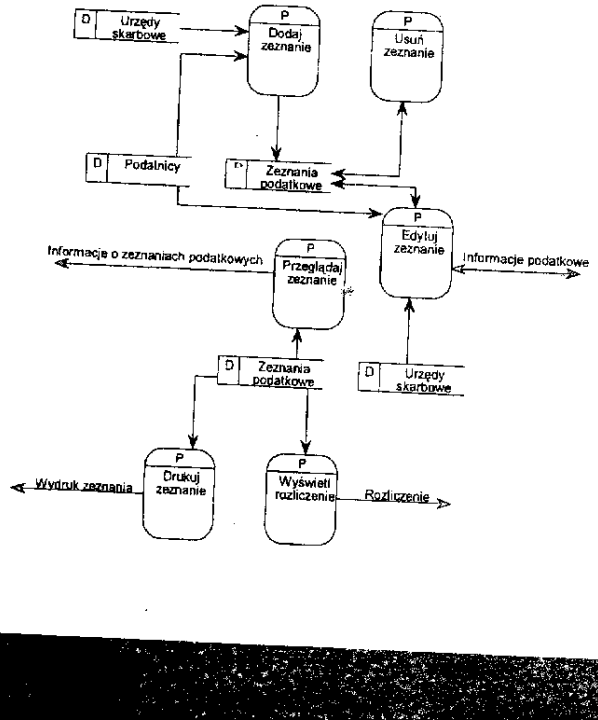
Rysunek 5.75. *Diagram kontekstowy systemu podatkowego*



Rysunek 5.76. *Diagram potomny procesu System podatkowy*



Rysunek 5.77. *Diagram potomny procesu Ewidencja zeznań podatkowych*



5.7.2. Podstawowe rezultaty fazy analizy

Podstawowe rezultaty fazy analizy to:

- poprawiony dokument opisujący wymagania,
- słownik danych wypełniony specyfikacją modelu,
- dokument opisujący stworzony model, który w przypadku stosowania metod obiektowych składa się z:
  - diagramu(ów) klas,
  - diagramów interakcji obiektów,
  - diagramów przejść stanów,
  - raportów:
    - definicji klas,
    - definicji pól,
    - definicji danych złożonych oraz elementarnych,
    - definicji metod,
- a w przypadku stosowania metod strukturalnych składa się z:
  - diagramu(ów) związków encji,
  - diagramów przepływów danych,
  - diagramów przejść stanów,
  - raportów:
    - definicji encji,
    - definicji atrybutów,
    - definicji procesów,
    - definicji zbiorników danych,
    - definicji przepływów danych,
    - definicji danych złożonych oraz elementarnych,
- harmonogram fazy projektowania.

5.7.3. Narzędzia CASE w fazie analizy

Jak wspomniano w sekcji 1.3 wstępnie omawiającej narzędzia CASE, systemy typu Upper-CASE koncentrują się głównie na fazie analizy. Także w pakietach typu I-CASE wspomaganie fazy analizy odgrywa bardzo istotną rolę.

Jednym z istotnych zadań narzędzi CASE w tej fazie jest wspomaganie posługiwania się notacjami graficznymi. Jednymi z podstawowych modułów tych narzędzi są więc edytory notacji graficznych stosowanych w fazie analizy. Diagramy graficzne mogą być oczywiście tworzone za pomocą edytorów graficznych ogólnego przeznaczenia, w praktyce byłoby to jednak zbyt niewygodne i czasochłonne. Edytory graficzne wchodzące

w skład narzędzi CASE są więc specjalizowane pod kątem poszczególnych notacji. Pozwalają one analitykowi skoncentrować się na jego najważniejszym zadaniu — analizie złożonego systemu.

Edytory graficzne powinny umożliwiać tworzenie powiązań pomiędzy pewnymi symbolami a diagramami. Przykłady takich powiązań, to:

- funkcja (wymaganie funkcjonalne) może być związana z diagramem interakcji opisującym sposób jej realizacji,
- akcje i operacje mogą być dekomponowane do diagramu przejść stanów,
- proces może być dekomponowany do diagramu przepływów danych,
- klasa lub encja może być związana z diagramem przejść stanów opisującym jej dynamiczne zachowanie,

Narzędzia CASE powinny umożliwiać także nawigowanie po diagramach z wykorzystaniem ich powiązań, to jest otwieranie diagramu związanego z wybranym symbolem oraz przechodzenie do diagramu zawierającego symbol związany z aktualnie edytowanym diagramem.

Kolejnym istotnym elementem narzędzi CASE wykorzystywanym w fazie analizy jest słownik danych. Służy on między innymi do przechowywania specyfikacji symboli pojawiających się na diagramach graficznych. W profesjonalnych narzędziach CASE słownik danych odgrywa kluczową rolę. Służy on do przechowywania praktycznie wszystkich informacji dotyczących tworzonego systemu, także tych zawartych na diagramach graficznych (poza być może informacjami o szczegółowym rozmieszczeniu symboli). W profesjonalnych narzędziach CASE diagramy graficzne są więc wygodnym narzędziem służącym wizualizacji i edycji informacji zawartych w słowniku danych. Słownik danych powinien pozwalać na wprowadzanie i edycję poszczególnych symboli także z pominięciem diagramów graficznych. W związku z tym wspomniane powyżej wiązanie pewnych symboli z diagramami dotyczy także symboli umieszczonych w słowniku danych lecz nie pojawiających się w danym momencie na żadnym diagramie. Słownik danych powinien pozwalać na wygodne wyszukiwanie potrzebnej informacji.

Ważne jest, aby narzędzia CASE zapewniały spójność informacji zawartej w słowniku danych oraz na różnych diagramach. Przykładowo informacja o metodach danej klasy znajdująca się w słowniku danych może być prezentowana jednocześnie na jednym lub kilku diagramach interakcji oraz diagramach klas. Zmiana, dodanie lub usunięcie metody powinny być dokonywane automatycznie (lub co najmniej po wydaniu polecenia odświeżenia) jednocześnie w słowniku danych i na wszystkich diagramach.

Narzędzia CASE powinny w miarę możliwości uniemożliwiać wprowadzanie błędnych konstrukcji, np. pętli w hierarchii generalizacji-specjalizacji, wiązania dwóch diagramów interakcji z jedną funkcją lub specyfikowania tej samej klasy w różny sposób. Nie wszystkie błędy można jednak wykryć na bieżąco. Niezbędne są więc specjalne funkcje służące do sprawdzania poprawności i kompletności modelu.

Jednym z wyników fazy analizy są raporty zawierające informacje umieszczone w słowniku danych. Narzędzia CASE powinny więc zawierać możliwości generowania rozmaitych raportów. Profesjonalne narzędzia pozwalają również na definiowanie własnych raportów.

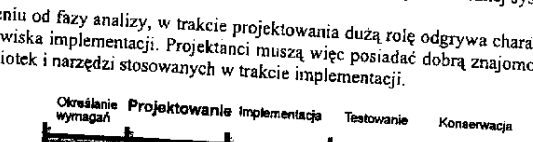
Dokumentacja fazy analizy może składać się z szeregu diagramów i raportów. Ich oddzielne drukowanie byłoby bardzo żmudne. Dlatego narzędzia CASE powinny ułatwiać definiowanie zawartości dokumentacji technicznej składającej się z różnych diagramów, raportów i opisów w języku naturalnym, która następnie będzie tworzona i uaktualniana automatycznie.

Rozdział 6. Projektowanie

Celem fazy projektowania jest opracowanie szczegółowego opisu implementacji systemu. Opis ten, po wykonaniu niezbędnych zmian wprowadzonych w trakcie implementacji oraz testowania, stanie się też główną składową dokumentacji technicznej systemu.

W odróżnieniu od fazy analizy, w trakcie projektowania dużą rolę odgrywa charakterystyka środowiska implementacji. Projektanci muszą więc posiadać dobrą znajomość języków, bibliotek i narzędzi stosowanych w trakcie implementacji.

Rysunek 6.1. Faza projektowania w cyklu życia oprogramowania



Jednym z podstawowych założeń inżynierii oprogramowania jest dążenie do tego, aby struktura tworzonego oprogramowania jak najbardziej zachowywała ogólną strukturę modelu stworzonego w poprzedniej fazie. Celem jest uzyskanie systemu, którego struktura odzwierciedlałaby naturalne zależności w danej dziedzinie problemu. Współczesne metody inżynierii oprogramowania są więc rozwinięciem znanej już od lat sześćdziesiątych idei programowania strukturalnego. Zachowanie przez system naturalnej struktury dziedziny problemu ma przede wszystkim podnieść jego zrozumiałość oraz ułatwić modyfikowanie systemu. Niewielkie zmiany w dziedzinie problemu powinny prowadzić do niewielkich zmian w oprogramowaniu, ich wprowadzenie nie powinno być więc zbyt kosztowne i długotrwałe. Niekorzystna jest sytuacja, w której klient żądający niewielkiej jego zdaniem zmiany, dowiaduje się, że będzie ona wymagała istotnych zmian w systemie, podważa to bowiem zaufanie klienta do producenta oprogramowania. Z drugiej strony klient łatwiej zaakceptuje konieczność dokonania istotnych zmian w systemie, jeżeli żądana przez niego modyfikacja jest poważna również z jego punktu widzenia. W związku z tym jednym z celów projektowania jest transformacja konstrukcji pojawiających się w modelu do odpowiednich konstrukcji środowiska implementacji. Ponieważ

jednak model budowany w fazie analizy abstrahuje od szczegółów środowiska implementacji, jest on zbyt ogólny, aby być bezpośrednią podstawą implementacji. Projektowanie wymaga więc uszczegółowienia wyników analizy.

Oprogramowanie składa się z kilku składowych. Jedną z nich jest składowa dziedziny problemu odpowiedzialna za wykonywanie podstawowych funkcji systemu oraz przechowywanie podstawowych dla systemu danych. Jest to składowa, która powstaje poprzez uszczegółowienie modelu zbudowanego w fazie analizy. W fazie projektowania należy opracować także inne składowe oprogramowania, tj.: składową kontaktu z użytkownikiem odpowiedzialną za współpracę systemu z użytkownikiem (użytkownikami), składową zarządzania danymi odpowiedzialną za przechowywanie oraz dostęp do trwałych danych, składową zarządzania pamięcią oraz składową zarządzania zadaniami.

W fazie analizy pewne funkcje i struktury danych mogą zostać zamodelowane w sposób mało efektywny. W fazie projektowania dokonywana jest więc również optymalizacja systemu.

Środowisko implementacji może zarówno narzucać pewne ograniczenia, to jest nie pozwalać na bezpośrednie zaimplementowanie pewnych konstrukcji pojawiających się w modelu, jak i oferować dodatkowe możliwości ułatwiające implementację. Projektowanie polega więc także na dostosowaniu systemu do ograniczeń i możliwości środowiska implementacji.

W fazie projektowania należy także określić fizyczną strukturę systemu.

Podsumowując, w fazie projektowania wykonywane są następujące zadania:

- uszczegółowienie wyników analizy,
- projektowanie składowych systemu nie związanych z dziedziną problemu,
- optymalizacja systemu,
- dostosowanie do ograniczeń i możliwości środowiska implementacji,
- określenie fizycznej struktury systemu.

6.1. Uszczegółowienie wyników analizy

Projekt musi być wystarczająco szczegółowy aby mógł być podstawą implementacji. Poziom szczegółowości projektu może więc zależeć od stopnia zaawansowania programistów odpowiedzialnych za implementację. Jeżeli są to osoby o dużym poziomie zaawansowania, będą w stanie wykonać implementację także na podstawie mniej szczegółowego projektu. Należy jednak pamiętać, że szczegółowy projekt będzie w przyszłości także dobrą dokumentacją techniczną.

Uszczegółowienie wyników analizy polega między innymi na wyborze sposobu implementacji pewnych konstrukcji pojawiających się w modelu, które mogą być zrealizowane na jeden z kilku sposobów. Firma programistyczna może jednak przyjąć pewne standardy, które jednoznacznie definiują sposób implementacji tego typu konstrukcji. Pozwala to w istotnym stopniu zredukować nakład pracy w fazie projektowania.

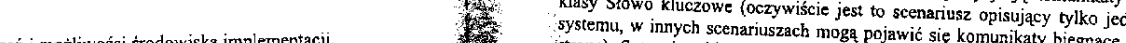
W przypadku metod strukturalnych w fazie projektowania i implementacji wykorzystywane są wyraźnie odmienne pojęcia niż te stosowane w fazie analizy. W związku z tym w metodach tych pojawiają się także notacje graficzne odmienne od stosowanych w fazie analizy. W przypadku metod obiektowych w fazach analizy, projektowania i implementacji wykorzystywane są te same pojęcia. We wszystkich fazach operuje się pojęciami klas, pól i metod. W związku z tym w projektowaniu obiektowym wykorzystywane są w zasadzie te same notacje graficzne co w fazie analizy, choć pojawiają się w nich pewne rozszerzenia stosowane tylko w fazie projektowania. Należy jednak dodać, że jest tak tylko w wypadku stosowania w fazie implementacji środowisk obiektowych.

6.1.1. Techniki obiektowe

Omówiona w sekcji 5.3 obiektowa notacja OMT może być wykorzystywana także w fazie projektowania. Zawiera ona dwie istotne konstrukcje wykorzystywane z reguły wyłącznie w fazie projektowania.

Związek skierowany (ang. *directed class link/association*)

Wprowadzony w sekcji 5.3 symbol związku klas (porównaj rysunek 5.8) traktuje obie klasy biorące udział w tym związku w sposób równoważny. Często jednak tylko obiekty jednej z powiązanych klas odwołują się do obiektów drugiej klasy. Przykładowo ze scenariusza przepływu komunikatów (rysunku 5.58) wynika, że w systemie informacji geograficznej tylko obiekty klasy Symbol graficzny wysyłają komunikaty do obiektów klasy Słowo kluczowe (oczywiście jest to scenariusz opisujący tylko jedną z funkcji systemu, w innych scenariuszach mogą pojawić się komunikaty biegnące w przeciwną stronę). Sytuacje takie pozwala opisać związek skierowany przedstawiony na rysunku 6.2.



Rysunek 6.2. Symbol związku skierowanego

Symbole poziomu dostępu do pól i metod

W typowych językach programowania istnieje możliwość ograniczania dostępu do pól i metod. W języku C++ istnieją np. trzy poziomy dostępności:

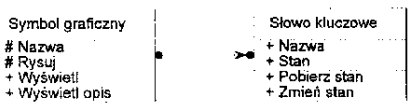
- dostęp publiczny oznacza, że dana składowa jest dostępna dla wszystkich funkcji i metod systemu mających dostęp do obiektów danej klasy,
- dostęp zabezpieczony (ang. protected access) oznacza, że dana składowa jest dostępna wyłącznie dla metod (oraz funkcji zaprzyjaźnionych) danej klasy oraz jej specjalizacji,

- dostęp prywatny oznacza, że dana składowa jest dostępna wyłącznie dla metod (oraz funkcji zaprzyjaźnionych) danej klasy.

O poziomie dostępności danej składowej informują dodatkowe symbole pojawiające się przed nazwą pola lub metody. Ich znaczenie jest następujące:

- „+" — dostęp publiczny,
- „#" — dostęp zabezpieczony,
- „–" — dostęp prywatny.

Przykład zastosowania tych symboli przedstawiono na rysunku 6.3.



Rysunek 6.3. Przykład zastosowania symboli poziomu dostępu do pól i metod

Dla metody Rysuj oraz pola Nazwa klasy Symbol graficzny wprowadzono poziom dostępu zabezpieczony, gdyż będą one wykorzystywane wyłącznie przez metody tej klasy i jej specjalizacji.

Notacją obiektową zorientowaną głównie na fazę projektowania jest notacja Boocha (Booch, 1994). Choć notacja ta nie zostanie dokładnie omówiona, warto wymienić pojawiające się w niej dodatkowe konstrukcje projektowe, które pozwalają opisywać:

- Wzorce klas (class templates).
- Metaklasy, tj. klasy zawierające pola i metody dotyczące klasy jako całości a nie pojedynczych obiektów, np. pola i metody statyczne.
- Wolne funkcje nie będące metodami żadnej z klas.
- Sposoby widoczności obiektu, do którego wysyłany jest komunikat. Obiekt ten może być widoczny gdyż znajduje się w tym samym zakresie leksykalnym, jest przekazany przez parametr lub jest polem klasy, której metoda wysyła komunikat.

W fazie projektowania w sposób bardziej szczegółowy opracowywana jest także specyfikacja projektu. Uszczegółowienie specyfikacji danych — pól, danych wejściowych i wyjściowych metod, danych elementarnych oraz złożonych struktur danych — polega przede wszystkim na określeniu ich typu w języku implementacji. W przypadku danych elementarnych należy wybrać typ elementarny języka implementacji odpowiadający wymaganiom określonym na etapie analizy. W przypadku danych złożonych konieczny jest wybór sposobu realizacji złożonych struktur danych. Poniżej podany jest przykład realizacji w językach C/C++ złożonej struktury danych. Jest to kontynuacja przykładu wprowadzonego w sekcji 5.3.4 omawiającej specyfikację modelu obiektowego.

Dane osobowe

Imię +
Nazwisko +
Adres

Dane osobowe składają z imienia nazwiska i adresu. Imię i nazwisko to dane elementarne. Adres to dana złożona zdefiniowana poniżej.

Adres

Ulica +
Numer domu +
(Numer mieszkania) +
Miasto +
Kod

Możliwa realizacja w C/C++

```
typedef struct {
            char Ulica [30];
            char NumerDomu [10];
            char NumerMieszkania [10];
            char Miasto [30];
            char Kod [5];
        } TAdres;

typedef struct {
            char Imie [20];
            char Nazwisko [30];
            TAdres Adres;
        } TDaneOsobowe;
```

Jeżeli język implementacji nie zawiera typów bezpośrednio odpowiadających potrzebom określonym w fazie analizy, możliwe jest zdefiniowane nowego typu jako klasy, w której zostaną wprowadzone operatory dla tego typu. Klasa taka nie staje się częścią modelu, lecz jest wygodnym sposobem implementacji typu o wymaganej charakterystyce. Analogiczna możliwość pojawia się oczywiście także w przypadku implementacji projektu strukturalnego w języku obiektowym.

Uszczegółowienie opisu metod wymaga szczegółowego opracowania ich nagłówków. Dane wejściowe powinny być przekazywane do metody przez jej parametry. Dane wyjściowe mogą być zwracane przez metodę za pomocą parametrów. Jeżeli metoda zostanie zrealizowana jako funkcja, może ona zwracać dane wyjściowe poprzez wartość tej funkcji. Jak wspomniano w sekcji 5.3.4 omawiającej specyfikację modelu obiektowego, algorytmy metod opracowuje się często dopiero na etapie projektowania w fazie analizy poprzestając na krótkim, jednozadaniowym opisie.

Na etapie analizy zakłada się, że wszystkie metody są metodami wirtualnymi. Z reguły nie jest to jednak wymagane dla większości metod. W fazie projektowania należy więc określić, które z metod będą zaimplementowane jako wirtualne.

Metody typu PobierzDane, UstawDane, PobierzPole, UstawPole realizuje się często jako bezpośrednie odwołania do poszczególnych pól.

Pola związane ograniczeniami z innymi polami w sposób, który pozwala obliczyć ich wartości na podstawie wartości innych pól, są często realizowane jako metody zwracające obliczane wartości wyeliminowanych pól. Przykładem może być pole Kwota dochodu, które jest związane następującym ograniczeniem:

Kwota dochodu = Kwota przychodu – Kwota kosztów

Pole to można więc zrealizować jako metodę zwracającą obliczoną wartość tego pola.

Kolejnym zadaniem wykonywanym w fazie projektowania jest określenie sposobu implementacji związków klas. Związki implementuje się wprowadzając do powiązanych klas dodatkowe pola. Pola te mogą być obiektami powiązanej klasy, wskaźnikami do obiektów powiązanej klasy lub identyfikatorami obiektów powiązanej klasy. Jeśli obiekty danej klasy związane są z nie więcej niż jednym obiektem drugiej klasy, wprowadzane pola mogą być pojedynczymi obiektami, wskaźnikami lub identyfikatorami. W przeciwnym wypadku konieczne jest stosowanie złożonych struktur danych pozwalających na przechowywanie zbiorów obiektów, wskaźników lub identyfikatorów. Jako przykład podane są możliwe sposoby realizacji związku przedstawionego na rysunku 6.3. Ponieważ obiekt klasy Symbol graficzny może być związany z wieloma obiektami klasy Słowo kluczowe, do klasy Symbol graficzny należy wprowadzić dodatkowe pole. Kilka możliwych sposobów realizacji podanych jest poniżej:

```
Slowo_kluczowe Slowa_kluczowe [100];
```

Tablica obiektów

```
list<Slowo_kluczowe*> Slowa_Kluczowe;
```

Lista wskaźników

```
char* Nazwy_slow_kluczowych [100];
```

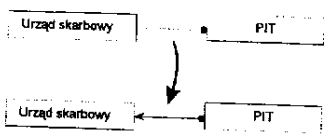
Tablica nazw (identyfikatorów)

Jeżeli implementacja jest wykonywana w języku obiektowym, to związki najczęściej realizuje się za pomocą pól będących wskaźnikami do obiektów. Pola będące obiektami stosuje się czasami do realizacji związków agregacji po stronie klasy będącej całością.

Jeżeli implementacja jest wykonywana w systemie relacyjnych baz danych, to związki realizuje się za pomocą pól będących identyfikatorami obiektów.

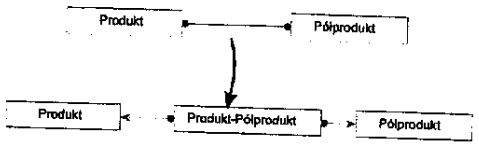
W przypadku związku skierowanego dodatkowe pola wprowadza się po stronie jednej z klas. W omawianym przykładzie z rysunku 6.3, pola wiążące obiekty zostałyby więc wprowadzone wyłącznie w klasie Symbol graficzny.

W pewnych środowiskach implementacji, przede wszystkim systemach relacyjnych baz danych, problemem może być implementacja dodatkowych pól, będących zbiorami identyfikatorów powiązanych obiektów ze względu na brak standardowych typów złożonych. Problem ten można czasami rozwiązać wprowadzając związki skierowane. Przykład takiego podejścia pokazany jest na rysunku 6.4 (porównaj sekcję 5.4.3.1, omawiającą analizę systemu podatkowego). Zgodnie z wprowadzonym na tym rysunku związkiem skierowanym wystarczające jest umieszczenie w klasie PIT pola będącego identyfikatorem pojedynczego obiektu klasy Urząd skarbowy. Transformacja tego typu może być jednak sprzeczna z określonym poprzednio przepływem komunikatów w systemie.



Rysunek 6.4. *Transformacja do związku skierowanego pozwalającego uniknąć wprowadzania pól*

Związki skierowane nie pozwalają jednak uniknąć problemów z implementacją dodatkowych pół będących zbiorami identyfikatorów powiązanych obiektów, jeżeli w związku może brać wiele obiektów obu klas. W tym przypadku wprowadza się dodatkowe klasy realizujące połączenie. Przykład tego podejścia przedstawiony jest na rysunku 6.5 (porównaj sekcję 5.4.3.3 omawiającą analizę systemu harmonogramowania zleceń).



Rysunek 6.5. *Dodatkowa klasa realizująca związek, w którym może brać wiele obiektów obu klas*

Związki ternarne implementuje się najczęściej poprzez wprowadzenie dodatkowej klasy związanej z wszystkimi klasami, które brały udział w oryginalnym związku ternarnym (porównaj rysunek 6.6).

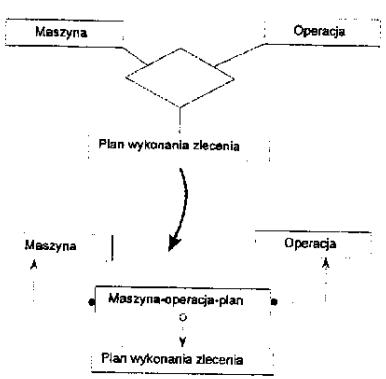
6.1.2. Techniki strukturalne

Podstawową notacją strukturalną wykorzystywaną w fazie projektowania są diagramy strukturalne (ang. *structure charts/diagrams*). Diagramy te posługują się odmiennymi pojęciami niż notacje stosowane w fazie implementacji. Pojęcia te oraz odpowiadające im symbole omówione są poniżej.

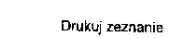
Moduł (ang. *module*)

Aktywna składowa programu, tj. procedura lub funkcja. Symbol modułu przedstawiony jest na rysunku 6.7.

Rysunek 6.6. *Realizacja związku ternarnego przez wprowadzenie dodatkowej klasy*



Rysunek 6.7. *Symbol modułu*



Moduł biblioteczny (ang. *library module*)

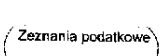
Jest to gotowa procedura lub funkcja wykorzystywana w systemie. Symbol modułu bibliotecznego przedstawiony jest na rysunku 6.8.

Rysunek 6.8. *Symbol modułu bibliotecznego*

Dane

Pod tym pojęciem rozumie się relację w bazie danych, plik lub zmienne programu. Symbol danych przedstawiony jest na rysunku 6.9.

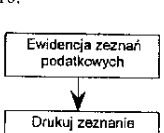
Rysunek 6.9. *Symbol danych*



Wywołanie (ang. *call*)

Pojęcie to opisuje wywoływanie przez pewien moduł innego modułu. Symbol wywołania przedstawiony jest na rysunku 6.10.

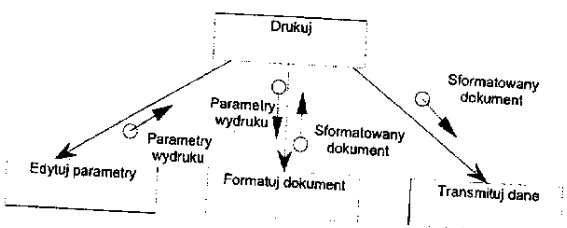
Rysunek 6.10. *Symbol wywołania*



Flagi przepływu danych (ang. *data flow flags*)

Z wywołaniem modułu może być związany przepływ danych z modułu wywołującego do wywoływanego i/lub z modułu wywoływanego do modułu wywołującego. Pierwszy rodzaj przepływu danych odpowiada parametrom wejściowym modułu wywoływanego, drugi rodzaj przepływu odpowiada parametrom wyjściowym. Symbole flag przepływu danych przedstawione są na rysunku 6.11.

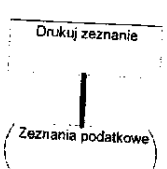
Rysunek 6.11. *Przykład diagramu strukturalnego z flagami przepływu danych*



Korzystanie z danych (ang. *data access*)

Symbol korzystania z danych (porównaj rysunek 6.12) oznacza, że moduł czyta, zapisuje lub modyfikuje dane. Z symbolem tym mogą być związane flagi przepływu danych.

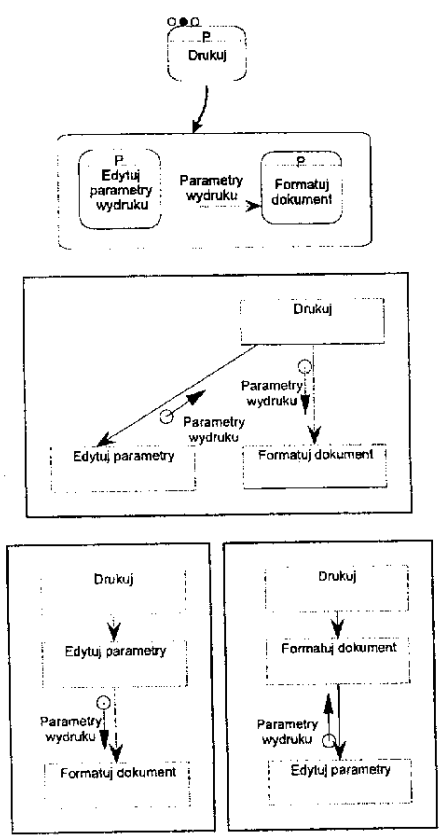
Rysunek 6.12. *Symbol korzystania z danych*



Diagramy strukturalne formatuje się z reguły tak, aby moduły wyższego poziomu — moduły wywołujące — znajdowały się powyżej modułów niższego poziomu — modułów wywoływanych (porównaj rysunek 6.11). Z reguły zakłada się, że moduły umieszczone z lewej strony wywoływane są wcześniej niż moduły umieszczone z prawej strony.

Diagramy strukturalne są uszczegółowieniem diagramów przepływów danych. Mogą być konstruowane poprzez transformacje istniejących diagramów przepływów danych. W tym wypadku procesy są transformowane do modułów. Diagramy strukturalne służą też do szczegółowego opisu procesów, które na etapie analizy zostały uznane za elementarne.

Diagramy przepływów danych nie zawierają informacji o sposobie wywołania poszczególnych procesów. Konstruując diagramy strukturalne na podstawie diagramów przepływów danych należy więc określić w jaki sposób moduły wzajemnie się wywołują. Jeżeli w modelu diagramów danych znajduje się proces, który jest dekomponowany do diagramu niższego poziomu, na który znajdują się procesy, pomiędzy którymi zachodzi przepływ danych, to możliwe są następujące rozwiązania (porównaj rysunek 6.13):



Rysunek 6.13. Możliwe sposoby transformacji diagramów przepływów danych do diagramu strukturalnego

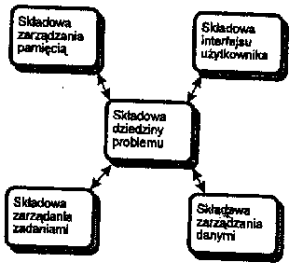
- moduł odpowiadający procesowi wyższego poziomu wywołuje najpierw moduł będący źródłem danych a następnie moduł będący odbiorcą danych,
- moduł odpowiadający procesowi wyższego poziomu wywołuje moduł będący źródłem danych, który z kolei wywołuje moduł będący odbiorcą danych,
- moduł odpowiadający procesowi wyższego poziomu wywołuje moduł będący odbiorcą danych, który z kolei wywołuje moduł będący źródłem danych.

Podobnie jak w przypadku projektowania obiektowego (porównaj punkt 6.1.1), uszczegółowieniu ulega też specyfikacja projektu. Opracowywane są nagłówki modułów oraz szczegółowa specyfikacja ich algorytmów. Precyzuje się też typy danych w wybranym środowisku implementacji.

6.2. Projektowanie składowych systemu nie związanych z dziedziną problemu

Projekt skonstruowany przez uszczegółowienie modelu opisuje składowe programu odpowiedzialne za realizację podstawowych zadań systemu. Gotowe oprogramowanie musi się jednak składać z dodatkowych składowych (porównaj rysunek 6.14):

- składowej interfejsu użytkownika, odpowiedzialnej za realizację komunikacji z użytkownikiem,
- składowej zarządzania danymi, odpowiedzialnej za realizację (na poziomie fizycznym) sposobu przechowywania trwałych danych,
- składowej zarządzania pamięcią operacyjną,
- składowej zarządzania zadaniami, odpowiedzialnej za przydział czasu procesora (procesorów) do poszczególnych zadań.



Rysunek 6.14. Składowe oprogramowania

W ostatnich latach obserwuje się gwałtowny rozwój narzędzi ułatwiających realizację składowych nie związanych z dziedziną problemu. Zarządzanie pamięcią oraz zadaniami jest z reguły wykonywane automatycznie przez kompilator oraz system operacyjny. Realizacja interfejsu użytkownika, szczególnie w środowiskach okienkowych, jeszcze do niedawna pochłaniała nawet 90% całkowitych nakładów. Obecnie nowoczesne systemy zarządzania interfejsem użytkownika (ang. *user interface management systems* — UIMS) pozwalają na budowę nawet bardzo złożonego interfejsu w sposób dialogowy praktycznie z pominięciem programowania. Do przechowywania trwałych danych najczęściej wykorzystuje się systemy baz danych.

Narzędzia szybkiego wytwarzania aplikacji (ang. *rapid application development* — RAD) pozwalają na łatwą realizację pewnych funkcji systemu poprzez tworzenie bezpośredniego połączenia pomiędzy elementami składowej kontaktu z użytkownikiem (dialogami, raportami) i elementami składowej zarządzania danymi (np. relacjami w relacyjnej bazie danych). W niektórych systemach można dzięki temu w sposób dialogowy zrealizować zdecydowaną większość funkcji. Mimo to, należy stwierdzić, że implementacja składowej dziedziny problemu w najmniejszym stopniu poddaje się automatyzacji. Wynika z tego rosnący udział nakładów przeznaczanych na realizację tej składowej.

Należy jednak pamiętać, że pewne systemy przeznaczone są do pracy w specyficznych środowiskach, w których mogą nie być dostępne systemy operacyjne ułatwiające zarządzanie pamięcią i zadaniami, systemy zarządzania bazami danych, czy środowiska ułatwiające realizację interfejsu użytkownika. W pewnych sytuacjach wymagania dotyczące efektywności i/lub ograniczenia zasobowe (np. niewielka pamięć operacyjna) mogą wymuszać rezygnację z narzędzi wysokiego poziomu. W takich wypadkach poszczególne składowe nie związane z dziedziną problemu należy konstruować w podobny sposób jak składową dziedziny problemu, rozpoczynając od budowy modelu, który następnie zostanie uszczegółowiony do projektu.

6.2.1. Projektowanie składowej kontaktu z użytkownikiem

Jak już wspomniano w ostatnich latach obserwuje się gwałtowny rozwój narzędzi ułatwiających realizację interfejsu użytkownika. Pojawienie się środowisk graficznych (na przykład MS Windows) umożliwiło co prawda tworzenie wygodnego w obsłudze interfejsu, początkowo wymagało jednak żmudnego programowania z wykorzystaniem bibliotek stosunkowo niskiego poziomu. Istotnym krokiem naprzód było wprowadzenie bibliotek obiektowych (na przykład Object Windows, Microsoft Foundation Class), których klasy grupowały w sobie możliwości wielu funkcji niskiego poziomu. Realizacja interfejsu użytkownika jest jednak najłatwiejsza z zastosowaniem systemów zarządzania interfejsem użytkownika (na przykład Zapp Factory, Visual Basic). Nowoczesne systemy tego typu pozwalają na:

- interaktywne projektowanie dialogów, okien, menu, map bitowych, ikon oraz pasków narzędziowych z wykorzystaniem bogatego zestawu gotowych elementów,
- definiowanie reakcji systemu na zajście pewnych zdarzeń, tj. akcji podejmowanych przez użytkownika (np. wybór opcji z menu),
- symulację pracy interfejsu,
- generowanie kodu, często z możliwością wyboru jednego z kilku środowisk docelowych.

W przypadku stosowania nowoczesnych narzędzi praca nad składową kontaktu z użytkownikiem koncentruje się na zaprojektowaniu wygodnego interfejsu, który jest realizowany przez automatycznie wygenerowany kod. Dlatego też w niniejszym rozdziale zostaną omówione zagadnienia organizacji komunikacji z użytkownikiem z pominięciem szczegółów implementacyjnych.

Można wyróżnić dwa podstawowe sposoby realizacji komunikacji z użytkownikiem:

- za pomocą linii komend,
- w pełnoekranowym środowisku okienkowym.

Ostatnio coraz częściej wykorzystywany jest ten drugi sposób. Nie oznacza to jednak, że ten rodzaj komunikacji z użytkownikiem jest zawsze lepszy. Tworzenie pełnoekranowego interfejsu ma sens w przypadku dużych systemów. Interfejs taki jest szczególnie wygodny dla początkujących i średnio zaawansowanych użytkowników. Zastosowanie linii komend często pozwala zaawansowanym użytkownikom na znacznie szybszą pracę niż interfejs pełnoekranowy. Linia komend wykorzystywana jest więc z reguły w przypadku niewielkich systemów oraz systemów wykorzystywanych przez zaawansowanych użytkowników.

Komunikacja użytkownika z systemem komputerowym polega na:

- wydawaniu przez użytkownika poleceń,
- przepływie danych pomiędzy użytkownikiem a systemem.

Typowe sposoby wydawania przez użytkownika poleceń systemowi to:

- wpisywanie poleceń za pomocą linii komend,
- wybór opcji z menu,
- wciśnięcie odpowiedniej kombinacji klawiszy (skrótu),
- korzystanie z ikon w paskach narzędziowych,
- wybór przycisku w dialogu,
- korzystanie z przycisków myszy.

Typowe sposoby wprowadzania danych to:

- podawanie parametrów poleceń w przypadku systemów z linią komend,
- wprowadzanie danych w odpowiedzi na zaproszenia systemu,
- wprowadzanie danych w dialogach.

Typowe sposoby przekazywania użytkownikowi danych to:

- wyświetlanie informacji w dialogach,
- wyświetlanie i/lub wydruki raportów,
- graficzna prezentacja danych.

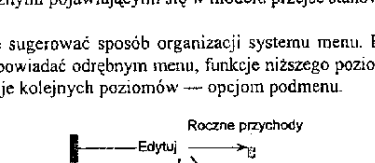
Prototyp interfejsu użytkownika może powstać już w fazie określania wymagań (porównaj sekcję 2.3 omawiającą prototypowanie). Systemy zarządzania interfejsem użytkownika pozwalają nie tylko na wygodną budowę takich prototypów, lecz także na wyko-

rzystanie prototypu w pełnym systemie po zintegrowaniu go ze składową dziedziny problemu. Jeżeli jednak nie zbudowano takiego prototypu, należy na podstawie modelu systemu określić wymagania wobec interfejsu użytkownika.

Polecenia wydawane przez użytkownika mogą być związane z następującymi elementami modelu:

- funkcjami (szczególnie elementarnymi) systemu,
- komunikatami wysyłanymi przez użytkownika do systemu pojawiającymi się w obiektowym modelu przepływu komunikatów,
- zdarzeniami zewnętrznymi pojawiającymi się w modelu przejść stanów.

Hierarchia funkcji może sugerować sposób organizacji systemu menu. Funkcje wysokiego poziomu mogą odpowiadać odrębnym menu, funkcje niższego poziomu — opcjom menu głównego, a funkcje kolejnych poziomów — opcjom podmenu.



Rysunek 6.15. Przykładowe sposoby realizacji komunikatów wysyłanych przez użytkownika

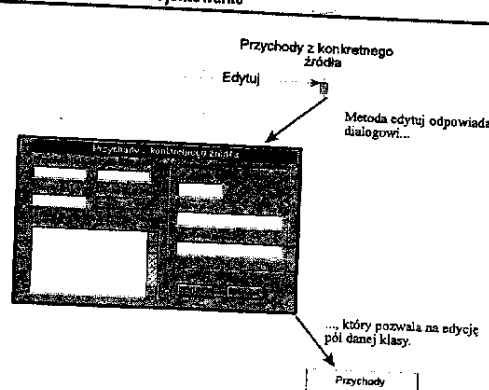
Wymiana danych pomiędzy użytkownikiem a systemem może być związana z następującymi elementami modelu:

- przepływami danych pomiędzy użytkownikiem a systemem,
- parametrami komunikatów wysyłanych przez użytkownika,
- metodami i procesami, które zgodnie ze specyfikacją służą do edycji obiektów, encji lub zbiorników danych.

Specyfikacja przepływów danych, parametrów, klas, encji lub zbiorników danych pozwala określić pola edycyjne, które należy umieścić w dialogach (patrz rysunek 6.16).

Oczywiście istnieje bardzo wiele sposobów realizacji interfejsu użytkownika zgodnego z wymaganiami wynikającymi z modelu. Niektóre z nich będą bardziej ergonomiczne niż inne. Zadaniem projektanta jest więc zaprojektowanie interfejsu użytkownika, który da możliwość korzystania z funkcji systemu w jak najwygodniejszy sposób. Istnieje kilka podstawowych zasad, zwanych złotymi zasadami, projektowania interfejsu użytkownika (porównaj Shneiderman 1993):

Rysunek 6.16. Dialog odpowiadający metodzie służącej do edycji obiektów danej klasy



- *Spójność.* Wygląd oraz obsługa systemu powinna być podobna w momencie korzystania z różnych funkcji. Oznacza to np., że poszczególne programy tworzące system powinny mieć zbliżony interfejs, podobnie powinna wyglądać praca z rozmaitymi dialogami, podobnie powinny być interpretowane operacje wykonywane za pomocą myszy. Chodzi tu między innymi o takie proste reguły jak:
  - umieszczanie etykiet zawsze nad lub obok pól edycyjnych,
  - umieszczanie typowych przycisków OK i Zaniechaj zawsze u dołu albo z prawej strony dialogu,
  - spójne tłumaczenie we wszystkich dialogach angielskiej nazwy przycisku Cancel, czyli wybór tylko jednego z możliwych tłumaczeń: Zaniechaj, Anuluj, Rezygnuj ...

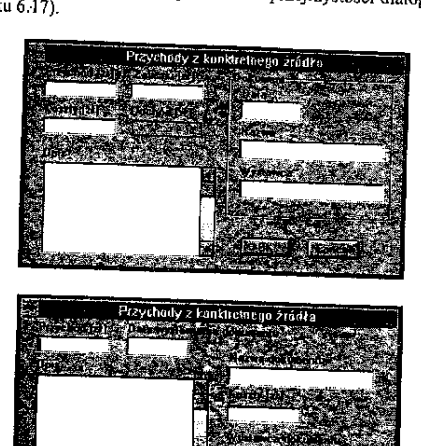
  Interfejs tworzonego systemu powinien być nie tylko wewnętrznie spójny, lecz również zgodny z innymi programami, z którymi styka się użytkownik.
- *Skróty dla doświadczonych użytkowników.* Korzystanie z menu i pasków narzędziowych jest wygodne dla początkujących użytkowników, którzy nie znają jeszcze dobrze możliwości systemu. Doświadczeni użytkownicy mogą jednak pracować zdecydowanie szybciej korzystając ze skrótów — kombinacji klawiszy pozwalających na natychmiastowe wydawanie poleceń.
- *Potwierdzanie przyjęcia polecenia użytkownika.* Realizacja pewnych poleceń może trwać stosunkowo długo. Jeżeli system nie informuje w żaden sposób użytkownika, że polecenie zostało przyjęte i jest realizowane, użytkownik może poczuć się zdezorientowany.

Pewien gracz giełdowy korzystając z terminala wydał polecenie sprzedaży pakietu akcji. Nacisnął klawisz Enter, kursor przeszedł do nowej linii i dość długo nic się nie działo. Zgodnie z jednym z praw Murphy'ego, które mówi, że każde urządzenie działa lepiej, jeśli kilka razy szybko naciśnie się uruchamiający je przycisk, gracz giełdowy kilkakrotnie nacisnął klawisz Enter. Po chwili uzyskał informację, że sprzedał właśnie nie jeden lecz pięć pakietów akcji.

- *Prosta obsługa błędów.* Jeżeli użytkownik wprowadzi w pewnym polu dialogu błędne dane (na przykład wykraczające poza dopuszczalny zakres), to po próbie zaakceptowanie wprowadzonych danych powinien nastąpić automatyczny powrót do tego dialogu z zachowaniem poprawnych wartości pól.
- *Odwoływanie akcji.* W najprostszym wypadku oznacza to możliwość odwołania ostatnio wykonanej akcji. Jeszcze lepiej jest jeżeli system pozwala odwoływać kolejno wiele akcji.
- *Wrażenie kontroli nad systemem.* Użytkownicy preferują systemy, które dają im pełne wrażenie kontroli. Deprymująco działają na użytkowników sytuacje, w których system sam rozpoczyna wykonywanie pewnych akcji lub nie pozwala przerwać rozpoczętych operacji.
- *Nieobciążanie pamięci krótkotrwałej.* Ludzka pamięć krótkotrwała ma stosunkowo ograniczoną pojemność (porównaj sekcję 13.1. omawiającą czynniki psychologiczne w inżynierii oprogramowania). W związku z tym projektant interfejsu użytkownika musi brać pod uwagę to, że użytkownik może zapomnieć jak i w jakim celu uruchomił pewien dialog. Tytuł dialogu lub pewne jego pola powinny więc przypominać o celu jego uruchomienia. Załóżmy że użytkownik przeglądający listę osób może uruchomić dialog edycji danych osobowych, w którym nie może jednak zmieniać nazwiska. Umieszczenie w tym dialogu nazwiska nie jest więc teoretycznie konieczne. Obciążałoby to jednak pamięć krótkotrwałą użytkownika, byłoby więc w praktyce poważnym błędem.
- *Grupowanie powiązanych operacji.* Jest to zasada związana z poprzednią. Wiele zadań, które będzie wykonywać użytkownik nie da się zamknąć w jednym prostym dialogu lub oknie. Wymagają one przejścia przez szereg dialogów, w których wprowadzane są różnorodne dane i wybierane różne opcje. W tym przypadku przydatne jest grupowanie dialogów i okien służących wykonaniu takiego zadania. Przykładem może być definiowanie nowego wykresu w arkuszu kalkulacyjnym. Microsoft Excel ułatwia to zadanie za pomocą „kreatora wykresów — narzędzia prowadzącego użytkownika przez kilka sekwencyjnych dialogów, pozwalającego w każdym momencie cofnąć się do wcześniejszych dialogów.

Wspomniana już psychologiczna reguła 7±2 (Miller, 1956), mówiąca że liczba obiektów, nad którymi człowiek jest w stanie jednocześnie pracować waha się między 5 a 9 powinna być brana pod uwagę także przy projektowaniu interfejsu użytkownika. Liczba opcji menu, podmenu lub pól w dialogu nie powinna więc przekraczać dziewięciu. Nowsze badania wskazują, że ograniczenie to można częściowo przełamać ułatwiając

użytkownikowi grupowanie obiektów. Przykładem może być wprowadzanie w menu separatorów oddzielających grupy podobnych poleceń lub grupowanie w dialogach powiązanych pól (czytelnik proszony jest o porównanie przejrzystości dialogów przedstawionych na rysunku 6.17).



Rysunek 6.17. *Dwa funkcjonalnie równoważne dialogi, z których tylko jeden uwzględnia wewnętrzne grupowanie pól*

6.2.2. Projektowanie składowej zarządzania danymi

Składowa zarządzania danymi obejmuje elementy systemu odpowiedzialne za przechowywanie danych trwałych, tj. trwałych klas, encji lub trwałych zbiorników danych. Poniżej omówione są pewne kluczowe decyzje dotyczące organizacji składowej zarządzania danymi.

Trwałe dane mogą być przechowywane:

- w zwykłych plikach systemu operacyjnego,
- w (najczęściej relacyjnej) bazie danych.

Poszczególne elementy danych — klasy lub encje — są najczęściej przechowywane:

- w jednej relacji lub pliku,
- w odrębnej relacji lub pliku dla każdej klasy lub encji.

Obiekty (wystąpienia klasy lub encji) mogą być sprowadzane do pamięci operacyjnej i trwale zapisywane:

- na bieżąco, kiedy konieczny jest dostęp do pewnego obiektu(ów) lub nastąpiła modyfikacja obiektu,
- na żądanie, kiedy użytkownik jawnie zleci zapis lub odczyt grupy obiektów.

Wymienione powyżej decyzje z reguły nie są niezależne. W przypadku systemów operujących na różnego rodzaju dokumentach (na przykład edytorów tekstowych i graficznych, arkuszy kalkulacyjnych), dokumenty składające się z wielu obiektów różnych klas są najczęściej zapisywane w jednym pliku systemu operacyjnego i odczytywane oraz zapisywane na żądanie użytkownika. Systemy operujące na dużym, globalnym zbiorze danych, z reguły korzystają z relacyjnych baz danych. Poszczególne encje lub klasy są przy tym zapisywane w oddzielnych relacjach. Obiekty są sprowadzane oraz zapisywane na bieżąco.

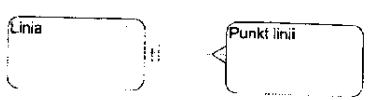
Zaletami relacyjnych baz danych są:

- wysoka efektywność i stabilność,
- możliwości zapewniania spójności danych poprzez automatyczne sprawdzanie warunków integralności,
- wielodostęp,
- rozszerzalność,
- możliwość rozproszenia danych,
- możliwość kaskadowego usuwania powiązanych obiektów.

Zaletą korzystania z plików systemu operacyjnego jest szybkość odczytu i zapisu dużej grupy powiązanych obiektów (na przykład obiektów tworzących dokument). Umieszczanie we wspólnej bazie danych grup różnorodnych obiektów tworzących odrębne, wzajemnie nie powiązane dokumenty prowadziłoby do znacznego wydłużenia czasów zapisu oraz odczytu danych.

Należy też uwzględnić fakt stałej długości rekordów w relacji. Pola klas lub atrybuty encji określone w fazie analizy mogą znacznie różnić się w wielkością w poszczególnych obiektach. Przykładem może być pole Lista punktów w klasie Linia systemu informacji geograficznej (porównaj sekcję 5.4.3.2 omawiającą analizę tego systemu). Liczba punktów definiujących poszczególne linie może być bardzo różna. Jedną z możliwości jest zaprojektowanie relacji o liczbie kolumn pozwalającej na przechowanie maksymalnej dopuszczalnej liczby punktów. Każdy rekord takiej relacji będzie jednak bardzo duży, prowadzi to więc do znacznego wzrostu wielkości zajmowanej pamięci.

Kolejne rozwiązanie polega na rozbiciu klasy Linia na dwie powiązane relacje przedstawione na rysunku 6.18. Odtworzenie klasy Linia wymaga jednak wykonania operacji połączenia tych relacji, wiąże się więc z dodatkowym narzutem czasu.



Rysunek 6.18. *Dodatkowa relacja Punkt linii jest sposobem realizacji pola o zmiennej długości*

Jeszcze jedna możliwość rozwiązania powyższego problemu polega na zastosowaniu pól o zmiennej długości, na przykład pól typu Memo. Rozwiązanie takie jest stosowane między innymi w pewnych narzędziach CASE do przechowywania zawartości słownika danych. Pola Memo są jednak polami tekstowymi. System musi samodzielnie zarządzać ich zawartością, traconych jest więc wiele zalet systemów relacyjnych baz danych.

Możliwość definiowania własnych typów danych jest jednym z rozszerzeń wprowadzanych obecnie w najnowszych systemach relacyjnych baz danych.

Projektowanie składowej zarządzania danymi w przypadku systemów operujących na dokumentach polega na:

- zaprojektowaniu formatu pliku,
- zaprojektowaniu scenariuszy odczytu oraz zapisu dokumentów oraz wprowadzeniu modułów lub metod odpowiedzialnych za te operacje.

Ważną sprawą jest zaprojektowanie sposobu zapisu oraz odczytu informacji o związkach obiektów. Należy także zapewnić obsługę błędów, które mogą wystąpić w trakcie odczytu lub zapisu.

W przypadku systemów operujących na dużym, globalnym zbiorze danych, projektowanie składowej zarządzania danymi polega na zaprojektowaniu relacyjnej bazy danych zgodnej z modelem opisującym trwałe dane oraz zaprojektowaniu zapytań wobec bazy danych. Na temat projektowania relacyjnych baz danych istnieje już stosunkowo bogata literatura w języku polskim (na przykład Pankowski, 1992). Podstawową notacją wykorzystywaną na tym etapie pozostają diagramy związków encji i klas. Symbole encji i klasy interpretowane są w tym wypadku jako odpowiedniki relacji w bazie danych. Pewne czynności związane z projektowaniem relacyjnych baz danych zostały już omówione w sekcji 6.1 poświęconej uszczegóławianiu wyników analizy. Były to:

- określenie sposobu realizacji związków pomiędzy klasami/encjami,
- określenie typów kolumn w relacjach odpowiadających polom/atrybutom klas/encji.

Projektując relacyjną bazę danych należy także:

- określić kolumny identyfikujące dla każdej relacji, mogą to być istniejące kolumny (grupy kolumn) lub sztuczne kolumny identyfikujące,
- znormalizować strukturę bazy danych,
- określić indeksy poszczególnych relacji.

6.3. Optymalizacja projektu

Implementacja projektu powstałego przez uszczegółowienie wyników analizy może prowadzić do systemu o zbyt niskiej efektywności, tzn.:

- wykonanie pewnych funkcji systemu może wymagać zbyt długiego czasu,
- struktury danych mogą wymagać zbyt dużej pamięci operacyjnej i masowej.

Dlatego jednym z zadań fazy projektowania jest poprawa efektywności systemu.

Optymalizacja może być realizowana:

- na poziomie projektu lub
- na poziomie implementacji.

Sposoby optymalizacji, które mogą być wykorzystywane na etapie implementacji to między innymi:

- stosowanie zmiennych statycznych zamiast dynamicznych i lokalnych,
- umieszczanie zagnieżdżonego kodu zamiast wywoływania procedur,
- dobór typów danych o minimalnej, niezbędnej wielkości,
- umieszczanie różnych danych w tych samych zmiennych (jeżeli zakres zmiennej pozwala na przechowywanie dwóch lub więcej danych).

Stosowanie powyższych technik jest oczywiście zadaniem programisty, projektant powinien określić jednak, które funkcje i struktury danych powinny zostać zoptymalizowane na etapie implementacji. Wiele kompilatorów posiada wbudowane mechanizmy optymalizacji pozwalające zredukować wpływ zastosowanych przez programistę konstrukcji, które mogłyby obniżyć efektywność systemu.

Optymalizacja na etapie implementacji rzadko jednak pozwala uzyskać wyraźną redukcję czasu obliczeń i zajętości pamięci. Zmiana algorytmu może natomiast w wielu przypadkach wielokrotnie zwiększyć efektywność czasową. Pozwala także często zmniejszyć tempo wzrostu czasu obliczeń przy wzroście rozmiaru problemu. Także przebudowa struktur danych może wielokrotnie zmniejszyć wymagania pamięciowe. Zagadnienia optymalizacji algorytmów i struktur danych są bardzo specyficzne dla konkretnych problemów (patrz np. Błażewicz, 1988).

Optymalizacja algorytmów oraz struktur danych może jednak prowadzić do znacznego zmniejszenia zrozumiałości i wzrostu stopnia komplikacji projektu. Może to być przyczyną dodatkowych błędów na etapie projektowania i implementacji oraz utrudnić dokonywanie modyfikacji w fazie eksploatacji. Dlatego też optymalizacja nie może być „sztuką dla sztuki". Powinna być stosowana wyłącznie w odniesieniu do funkcji, które rzeczywiście realizowane są w zbyt długim czasie oraz do struktur danych, które powodują rzeczywiste problemy z pamięcią operacyjną i masową. Redukcja czasu realizacji funkcji wykonywanej jednorazowo na żądanie użytkownika z 0.1 sekundy do 0.01 sekundy będzie dla niego zupełnie niezauważalna. Nawet w dużych systemach często tylko kilka funkcji jest krytycznych ze względu na czas realizacji. Znane jest twierdzenie, że 10% kodu jest wykonywane przez 90% czasu pracy systemu. Podobnie wygląda wpływ poszczególnych struktur danych na zajętość pamięci.

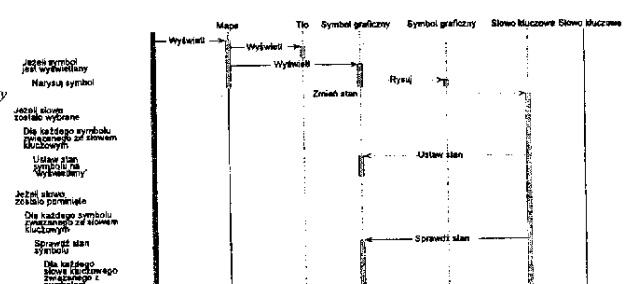
Wykrycie krytycznych funkcji wymaga dobrej znajomości środowiska implementacji. Może też wymagać przeprowadzenia testów obliczeniowych, które pozwolą określić czas realizacji poszczególnych funkcji.

Na efektywność wykonywania funkcji systemu duży wpływ może mieć sposób realizacji związków klas i encji. Zastosowanie pól będących wskaźnikami do powiązanych obiektów zamiast ich identyfikatorów może znacznie skrócić czas dostępu. Wskaźniki nie mogą być jednak zapisywane w pamięci masowej. Stosowanie wskaźników komplikuje więc sposób realizacji składowej zarządzania danymi.

W przypadku systemu informacji geograficznej (porównaj sekcję 5.4.3.2 omawiającą analizę tego systemu) funkcją krytyczną ze względu na czas jest wyświetlanie mapy. Jednym z elementów optymalizacji tej funkcji jest zaprojektowanie sposobu określania czy dany symbol graficzny powinien być wyświetlany — czy leży w wyświetlanym obszarze i czy jest opisywany przez przynajmniej jedno aktualnie wybrane przez użytkownika słowo kluczowe. Jeżeli zakładamy, że liczba słów kluczowych może być duża, to dobrym rozwiązaniem będzie sprawdzanie najpierw czy dany symbol leży (przynajmniej częściowo) w wyświetlanym obszarze mapy, a dopiero później sprawdzanie stanu powiązanych z nim słów kluczowych.

Sprawdzenie stanu słów kluczowych opisujących dany symbol graficzny będzie znacznie szybsze jeżeli związek pomiędzy tymi klasami zostanie zrealizowany za pomocą wskaźników. Jeżeli obiekt klasy Symbol graficzny będzie przechowywał listę wskaźników do opisujących go obiektów klasy Słowo kluczowe, to sprawdzenie czy powinien on być wyświetlany będzie wymagało przejrzenia tylko tych słów kluczowych. Zastosowanie wskaźników skomplikuje jednak realizację operacji zapisu i odczytu mapy.

Jeżeli spodziewamy się, że użytkownik często będzie wywoływał funkcję wyświetlania mapy, np. zmieniając stopień powiększenia lub przewijając zawartość okna zawierającego mapę, to efektywne może być wprowadzenie dodatkowego pola w klasie Symbol graficzny opisującego stan obiektów tej klasy. Pole to byłoby ustawiane we wszystkich niezbędnych obiektach klasy Symbol graficzny po zmianie stanu dowolnego słowa kluczowego. Jeżeli żadne ze słów kluczowych, z którym związany byłby dany symbol nie byłoby wybrane, to dodatkowo wprowadzone pole informowałoby, że symbol nie powinien być wyświetlany. W trakcie wyświetlania mapy nie byłoby konieczne odwoływanie się do słów kluczowych związanych z danym symbolem graficznych. Czasochłonna byłaby jednak operacja wykonywana po zmianie stanu dowolnego słowa kluczowego, zwłaszcza po pominięciu słowa kluczowego. Podejście to nie byłoby więc sensowne, gdyby użytkownik często zmieniał stan słów kluczowych. Opisane rozwiązanie odpowiada diagramowi interakcji przedstawionemu na rysunku 6.19.



Rysunek 6.19. Zoptymalizowany scenariusz przeglądania mapy

6.4. Dostosowanie do ograniczeń i możliwości środowiska implementacji

Środowisko implementacji systemu może zarówno nie pozwalać na bezpośrednią implementację pewnych konstrukcji modelu, jak również pozwalać na stosunkowo łatwą implementację pewnych elementów modelu dzięki możliwości wykorzystania gotowych składowych.

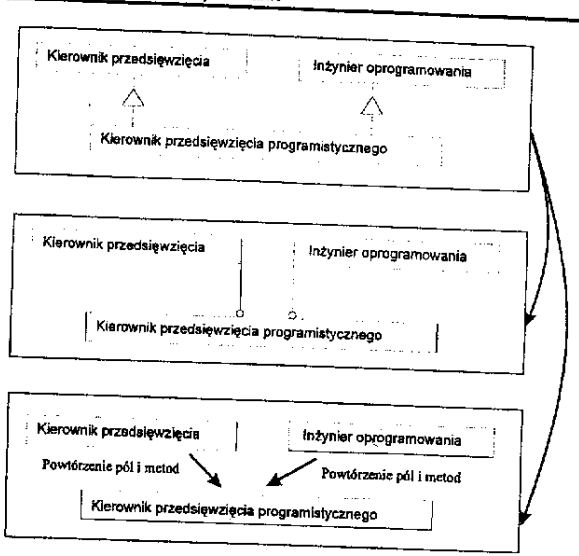
Typowe ograniczenia środowisk implementacji, z którym może zetknąć się projektant to:

- brak dziedziczenia wielokrotnego,
- brak dziedziczenia,
- brak metod wirtualnych.

Wszystkie te ograniczenia należy brać pod uwagę w przypadku implementacji z wykorzystaniem systemów relacyjnych baz danych i języków proceduralnych.

Brak dziedziczenia wielokrotnego

Jednym z rozwiązań jest zastąpienie związku generalizacji-specjalizacji związkiem klas lub encji (porównaj rysunek 6.20). Inne rozwiązanie polega na umieszczeniu wszystkich pól i metod obu generalizacji w specjalizacji (porównaj rysunek 6.20), czyli innymi słowy rezygnacja z dziedziczenia wielokrotnego.



Rysunek 6.20. Sposoby omijania braku dziedziczenia wielokrotnego

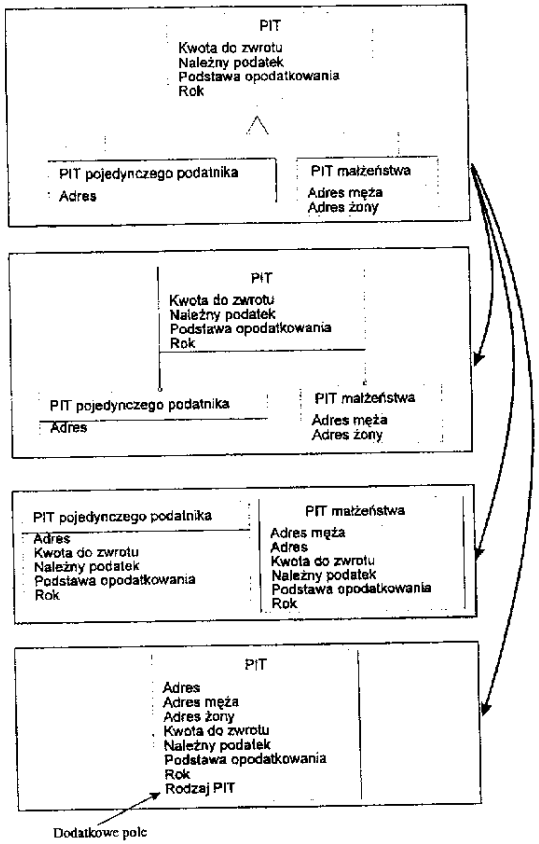
Brak dziedziczenia

Jeżeli środowisko programistyczne w ogóle nie zawiera mechanizmów dziedziczenia, to możliwe są następujące rozwiązania (porównaj rysunek 6.21):

- Zastąpienie związku generalizacji-specjalizacji związkiem klas lub encji.
- Umieszczenie wszystkich pól i metod generalizacji na poziomie specjalizacji.
- Umieszczenie na poziomie generalizacji pól i metod wszystkich jej specjalizacji oraz dodatkowego pola opisującego rodzaj aktualnego obiektu. Rozwiązanie to prowadzi oczywiście do zwiększonej zajętości pamięci, jest jednak jedynym, które zapewnia zgodność typu specjalizacji z typem generalizacji. Jest też jedynym, które pozwala wprowadzić pewien rodzaj metod wirtualnych.

Brak metod wirtualnych

Jednym ze sposobów omijania braku metod wirtualnych jest unikanie ich stosowania. Oznacza to konieczność jawnego wysyłania komunikatów do obiektów, których metody są przez te komunikaty wywoływane. Inaczej mówiąc projektant byłby odpowiedzialny za wykonanie transformacji przedstawionej na rysunku 5.45. Innym sposobem jest wprowadzenie na poziomie generalizacji dodatkowego pola identyfikującego specjalizację, do której należy dany obiekt. Przykładowo w sytuacji przedstawionej na rysunku 5.45 do klasy Figura można wprowadzić pole Rodzaj figury, które przyjmowałoby wartości „Linia" lub „Elipsa".



Rysunek 6.21. Sposoby omijania braku dziedziczenia

Metoda Wyświetl klasy Figura miałaby następujący algorytm:

```
W zależności od wartości pola Rodzaj figury
    Jeżeli pole ma wartość „Linia", to
        Wywołaj metodę Wyświetl klasy Linia
    Jeżeli pole ma wartość „Elipsa", to
        Wywołaj metodę Wyświetl klasy Elipsa
```


Inaczej mówiąc, rozwiązanie polega na wykonaniu przez projektanta i programistę czynności, które są wykonywane automatycznie przez obiektowe języki programowania.

Środowisko implementacji może zawierać szereg gotowych składowych, które mogą ułatwić implementację. Może się okazać, że biblioteki, które standardowo obejmuje środowisko implementacji, lub które można dodatkowo nabyć, obejmują funkcje i klasy o możliwościach bardzo zbliżonych do klas funkcji i klas wprowadzonych w modelu. Oczywiście dość rzadko zdarza się, że składowe biblioteczne dokładnie odpowiadają składowym modelu. Wykorzystanie składowych bibliotecznych może więc wymagać:

- zmodyfikowania składowych bibliotecznych (na przykład poprzez wprowadzenie specjalizacji klas bibliotecznych),
- dostosowanie składowej dziedziny problemu do wymogów składowych bibliotecznych.

To drugie rozwiązanie może oczywiście zaburzyć naturalną strukturę składowej dziedziny problemu. Projektant powinien więc znaleźć właściwy kompromis pomiędzy dążeniem do zachowania struktury modelu, a dążeniem do osiągnięcia korzyści związanych ze stosowaniem gotowych składowych (porównaj sekcję 2.6. omawiającą montaż z gotowych komponentów).

6.5. Określenie fizycznej struktury systemu

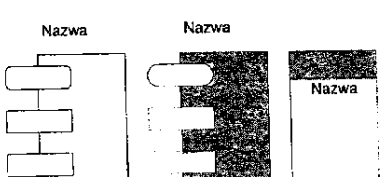
Określenie fizycznej struktury systemu obejmuje:

- Określenie struktury kodu źródłowego, to jest wyróżnienie plików źródłowych, zależności pomiędzy nimi oraz rozmieszczenie składowych projektu w plikach źródłowych.
- Podział systemu na poszczególne aplikacje.
- Fizyczne rozmieszczenie danych i aplikacji na stacjach roboczych i serwerach.

Jako narzędzia wspomagające projektowanie struktury kodu źródłowego proponuje się między innymi odrębne notacje graficzne. Przykładem może być omówiona poniżej, wprowadzona przez Boocha (1994) notacja, dostosowana szczególnie do potrzeb języka C/C++.

Deklaracja

Jest to symbol odpowiadający plikowi zawierającemu deklaracje (na przykład deklaracje klas i funkcji). Może opisywać na przykład plik nagłówkowy języka C/C++. Symbol deklaracji przedstawiony jest na rysunku 6.22.a.



Rysunek 6.22.
Symbole:
a. deklaracji
b. definicji
c. modułu głównego

Definicja

Jest to symbol odpowiadający plikowi zawierającemu definicje (np. definicje klas i funkcji). Może opisywać np. plik definicyjny języka C/C++. Symbol definicji przedstawiony jest na rysunku 6.22.b.

Moduł główny

Jest to symbol odpowiadający głównemu modułowi pewnej aplikacji. Może na przykład opisywać plik zawierający funkcję main języka C/C++. Wprowadzenie jednego lub kilku takich symboli oznacza więc jednocześnie wyróżnienie jednej lub kilku aplikacji wchodzących w skład systemu. Symbol modułu głównego przedstawiony jest na rysunku 6.22.c.

Zależność kompilacji

Symbol ten oznacza konieczność rekompilacji danego pliku po zmianie innego.

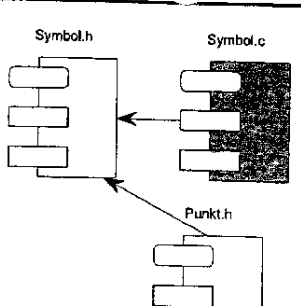
Moduły specyfikuje się podając listę składowych projektu, które są w nich umieszczone. W przypadku projektu obiektowego jest to np. lista klas deklarowanych i definiowanych w danym module.

Duży system może składać się z wielu odrębnych aplikacji. Wyróżniając aplikacje projektant powinien kierować się opisem wymagań funkcjonalnych, skonstruowanym w fazie określania wymagań. Poszczególne programy powinny odpowiadać konkretnym funkcjom lub grupom funkcji najczęściej wysokiego poziomu. Funkcje wchodzące w skład danej aplikacji powinny być również funkcjami wykorzystywanymi przez konkretną klasę użytkowników.

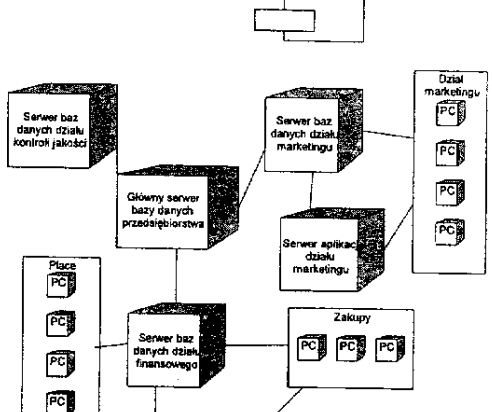
W przypadku niewielkiego systemu wszystkie dane mogą być przechowywane, a wszystkie aplikacje uruchamiane na jednym komputerze. Duże systemy mogą obejmować:

- jeden lub wiele serwerów baz danych,
- jeden lub wiele serwerów aplikacji,
- wiele stacji roboczych, to jest komputerów wykorzystywanych przez końcowych użytkowników.

Przykład prostej notacji zaproponowanej przez Boocha (1994) do opisu sprzętowej konfiguracji systemu podany jest na rysunku 6.24.



Rysunek 6.23.
Symbole zależności kompilacji



Rysunek 6.24.
Graficzny opis sprzętowej konfiguracji systemu

6.6. Poprawność projektu

Poprawność projektu oznacza, że jego opis jest zgodny z zasadami posługiwania się notacjami wykorzystywanymi do jego zapisu. Poprawność projektu nie gwarantuje więc, że opisuje on system, który rzeczywiście będzie zgodny z wymaganiami użytkownika. Jest to więc sytuacja analogiczna do kodu programu wolnego od błędów kompilacji, który może jednak nie działać poprawnie.

Poprawny projekt musi być:

- *kompletny*,
- *niesprzeczny*,
- *spójny*,
- *zgodny z regułami składniowymi notacji*.

Kompletność projektu oznacza, że wszystkie elementy są zdefiniowane w należyty sposób. W przypadku projektu obiektowego oznacza to, że zdefiniowane są:

- wszystkie klasy,
- wszystkie pola,
- wszystkie metody,
- wszystkie dane złożone i elementarne,

a także, że opisany jest sposób realizacji wszystkich wymagań funkcjonalnych.

Niesprzeczność oznacza, że żaden element nie ma dwóch i więcej sprzecznych definicji.

Spójność oznacza semantyczną zgodność informacji zawartych na różnorodnych diagramach i w specyfikacji.

Spójność oraz zgodność z regułami składniowymi zależy oczywiście silnie od rodzaju stosowanej notacji. Dlatego w dalszej części zostaną omówione typowe reguły poprawności opisanych w poprzednich rozdziałach notacji.

Diagramy klas

Warunki poprawności diagramów tego typu to (porównaj sekcję 5.3.1 omawiającą ten rodzaj diagramów):

- acykliczność związków generalizacji-specjalizacji,
- opcjonalność cyklicznych związków klas i związków agregacji,
- brak klas nie powiązanych w żaden sposób z innymi klasami (sytuacja ta może się pojawić, jeżeli projekt opisuje bibliotekę a nie gotowy system),
- umieszczenie w specyfikacji metod informacji o odczycie danych wejściowych oraz o ustawianiu danych wyjściowych.

Diagramy interakcji

Warunki poprawności diagramów tego typu to (porównaj sekcję 5.3.2 omawiającą ten rodzaj diagramów):

- istnienie metod wywoływanych przez komunikaty w klasach, do których wysyłane są te komunikaty lub w ich generalizacjach,

- umieszczenie w specyfikacji metod, które zgodnie z diagramem wysyłają komunikaty, informacji o wysyłaniu tych komunikatów.

Diagramy przejść stanów

Warunki poprawności diagramów tego typu to (porównaj sekcję 5.3.3 omawiającą ten rodzaj diagramów):

- brak stanów (z wyjątkiem startowego), do których nie ma możliwości przejścia,
- brak stanów (z wyjątkiem końcowego), z których nie ma możliwości wyjścia,
- jednoznaczność przejść ze stanów pod wpływem poszczególnych zdarzeń.

Diagramy związków encji

Warunki poprawności diagramów tego typu to (porównaj sekcję 5.5.1 omawiającą ten rodzaj diagramów):

- acykliczność związków generalizacji-specjalizacji,
- opcjonalność cyklicznych związków encji,
- brak encji nie powiązanych w żaden sposób z innymi encjami.

Diagramy przepływów danych

Warunki poprawności diagramów tego typu to (porównaj sekcję 5.5.2 omawiającą ten rodzaj diagramów):

- brak przepływów danych pomiędzy zbiornikami danych,
- brak przepływów danych między systemami zewnętrznymi a zbiornikami danych,
- zgodność specyfikacji przepływów danych wpływających i wypływających do poszczególnych procesów posiadających diagramy potomne z przepływami danych umieszczonymi na tych diagramach,
- umieszczenie w specyfikacji procesów informacji o odczycie oraz ustawianiu danych, zgodnych z przepływami wpływającymi i wypływającymi z poszczególnych procesów.

Diagramy strukturalne

Warunki poprawności diagramów tego typu to (porównaj sekcję 6.1.2 omawiającą ten rodzaj diagramów):

- brak modułów nie wywoływanych przez inne moduły (z wyjątkiem głównego),
- umieszczenie w specyfikacji modułów informacji o odczycie danych wejściowych i ustawianiu danych wyjściowych,
- umieszczenie w specyfikacji modułów informacji o wywoływaniu modułów niższego poziomu.

6.7. Jakość projektu

Omówione w poprzednich rozdziałach metody analizy i projektowania nie są ścisłymi algorytmami. Różne osoby posługując się podobnymi notacjami i metodami mogą dla tego samego problemu skonstruować różne, choć w pełni poprawne modele i projekty. Powstaje więc pytanie, w jaki sposób można ocenić jakość danego projektu?

W sekcji 1.2, omawiającej zakres inżynierii oprogramowania, wymieniono ogólne kryteria jakości oprogramowania: zgodność z wymaganiami użytkownika, niezawodność, efektywność, łatwość konserwacji i ergonomiczność. Kryteria te są jednak zbyt ogólne, aby mogły być stosowane na etapie analizy i projektowania. Proponuje się więc bardziej szczegółowe kryteria, które można odnieść bezpośrednio do modelu i projektu. Są to:

- *spójność*,
- *stopień powiązań składowych*,
- *przejrzystość*.

6.7.1. Spójność

Analiza i projektowanie polega między innymi na dzieleniu złożonych elementów na części składowe oraz grupowaniu składowych w większe całości. Przykładowo:

- klasa składa się z pól i metod, encja składa się z atrybutów,
- encje i klasy mogą składać się z encji lub klas będących częściami składowymi (związek agregacji),
- procesy mogą składać się z podprocesów.

Spójność opisuje na ile poszczególne składowe „pasują" do siebie. Jeżeli poszczególne składowe większej całości będą ze sobą spójne, to jest duża szansa, że przyszłe zmiany systemu będą wymagały modyfikacji ściśle określonej części systemu.

Constantine i Yourdon (1979) wyróżniają kilka poziomów spójności. Poziomy te zostały zaproponowane z myślą o metodach strukturalnych, w szczególności grupowaniu procesów i modułów. Pewne poziomy spójności mogą zostać jednak odniesione także do modelu i projektu obiektowego. Proponowane poziomy spójności, uszeregowane od najniższego do najwyższego, to:

- *Spójność przypadkowa*. Powstaje np. wtedy, gdy programista dzieli program na moduły tylko dlatego, że umieszczenie wszystkich funkcji w jednym dużym pliku utrudniało lub uniemożliwiało jego edycję.
- *Spójność logiczna*. Poszczególne składowe realizują podobne funkcje, np. obsługę błędów, wykonywanie podobnych obliczeń.

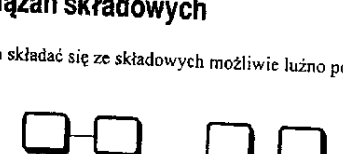
- *Spójność czasowa*. Składowe są uruchamiane/wykorzystywane w podobnym czasie, np. przy starcie lub końcu pracy systemu.
- *Spójność proceduralna*. Składowe tworzą sekwencję wykonania, czyli są kolejno uruchamiane.
- *Spójność komunikacyjna*. Składowe operują wspólnie na tym samym zestawie danych wejściowych i wspólnie produkują ten sam zestaw danych wyjściowych.
- *Spójność sekwencyjna*. Dane wyjściowe jednej składowej są danymi wejściowymi kolejnej składowej.
- *Spójność funkcjonalna*. Wszystkie składowe są niezbędne dla wykonania pewnej, dającej się logicznie wyróżnić funkcji.

Definicje powyższych poziomów spójności nie są w pełni precyzyjne. W praktyce nie zawsze jest więc jasne jak należy ocenić daną składową, a oceny różnych osób mogą być różne. W przypadku modelu i projektu obiektowego definiuje się tak zwaną spójność obiektową. Ten rodzaj spójności występuje wtedy, gdy pola danej klasy opisują dający się logicznie wyróżnić byt a jej metody ustawiają, modyfikują i odczytują te pola.

Z powyższego opisu wynika, że budowa modelu zgodna z omówionymi wcześniej zasadami, a potem jego transformacja do projektu dziedziny problemu, powinny zapewnić osiągnięcie najwyższych poziomów spójności, tj. spójności funkcjonalnej i obiektowej.

6.7.2. Stopień powiązań składowych

Dobry projekt powinien składać się ze składowych możliwie luźno powiązanych pomiędzy sobą.



Rysunek 6.25. Przykłady systemów ściśle i luźno powiązanych

Wynika to z faktu, że modyfikacja pewnej składowej może wymuszać zmiany w powiązanych z nią składowych. Przykładowo modyfikacja pewnej struktury danych wymusza zmiany we wszystkich modułach, które z tej struktury korzystają.

W przypadku stosowania technik strukturalnych powiązania składowych to:

- korzystanie przez procesy/moduły z tych samych danych,
- przepływy danych pomiędzy procesami/modułami,
- związki encji.

W przypadku stosowania technik obiektowych powiązania składowych to:

- przepływy komunikatów,
- dziedziczenie,
- związki klas.

Stopień powiązań składowych można mierzyć za pomocą pewnych miar liczbowych. Przykłady takich miar to np. średnie liczby komunikatów wysyłanych/otrzymywanych przez pojedynczą klasę.

6.7.3. Przejrzystość

Dobry projekt powinien być przejrzysty, czyli czytelny, łatwo zrozumiały. Na przejrzystość projektu wpływają następujące czynniki:

- *Odzwierciedlanie rzeczywistości.* Składowe i ich związki pojawiające się w projekcie powinny odzwierciedlać strukturę problemu. Ponownie należy stwierdzić, że poprawna budowa modelu opisującego dziedzinę problemu, a następnie jego transformacja do projektu, zapewniają ścisły związek projektu z rzeczywistością.
- *Spójność oraz stopień powiązań składowych.* Omówione powyżej kryteria oceny jakości projektu wpływają także na jego przejrzystość. Spójne i luźno powiązane składowe są zrozumiałe bez odwoływania się do innych składowych.
- *Zrozumiałe nazewnictwo.*
- *Czytelna i pełna specyfikacja.*
- *Odpowiednia złożoność składowych.*

Na uwagę zasługuje wpływ dziedziczenia na przejrzystość projektu. Z jednej strony definiowanie nowej klasy na bazie innej (innych) klas(y) pozwala znacznie uprościć jej definicję, wpływa więc na wzrost przejrzystości. Z drugiej strony pełne zrozumienie danej klasy jest niemożliwe bez przeanalizowana wszystkich klas bazowych.

6.8. Podsumowanie

6.8.1. Kluczowe czynniki sukcesu

Najistotniejsze elementy wpływające na sukces fazy projektowania to:

- *Wysoka jakość modelu.*
- *Dobra znajomość środowiska implementacji.*

- *Zachowanie przyjętych standardów* dotyczących np. stosowanych notacji.
- *Sprawdzenie poprawności projektu.*
- *Optymalizacja projektu we właściwym zakresie.* Optymalizacja powinna być ograniczona wyłącznie do tych fragmentów systemu, gdzie jest ona rzeczywiście konieczna.

6.8.2. Podstawowe rezultaty fazy projektowania

Podstawowe rezultaty fazy projektowania to:

- poprawiony dokument opisujący wymagania,
- poprawiony model,
- uzupełniona i uszczegółowiona specyfikacja projektu zawarta w słowniku danych,
- dokument opisujący stworzony projekt, który w przypadku stosowania metod obiektowych składa się z:
  - diagramu(ów) klas,
  - diagramów interakcji obiektów,
  - diagramów przejść stanów,
  - innych diagramów o charakterze projektowym, np. diagramów Boocha,
  - raportów:
    - definicji klas,
    - definicji pól,
    - definicji danych złożonych oraz elementarnych,
    - definicji metod,

  a w przypadku stosowania metod strukturalnych z:
  - diagramu(ów) związków encji,
  - diagramów przepływów danych,
  - diagramów przejść stanów,
  - diagramów strukturalnych,
  - raportów:
    - definicji encji,
    - definicji atrybutów,
    - definicji procesów,
    - definicji zbiorników danych,
    - definicji przepływów danych,
    - definicji danych złożonych oraz elementarnych,
    - definicji modułów,

- zasoby interfejsu użytkownika, np. menu, dialogi...,
- projekt bazy danych,
- projekt fizycznej struktury systemu,
- poprawiony plan testów,
- harmonogram fazy implementacji.

6.8.3. Narzędzia CASE w fazie projektowania

Tradycyjnie na wspomaganiu prac w fazie projektowania koncentrowały się narzędzia typu Lower-CASE. Narzędzia te nie są niezależne od konkretnych środowisk implementacji.

W fazie projektowania wykorzystuje się podobne składowe narzędzi CASE, do tych stosowanych w fazie analizy, czyli:

- edytory notacji graficznych,
- narzędzia edycji słownika danych,
- generatory raportów,
- generatory dokumentacji technicznej,
- narzędzia sprawdzania jakości projektu.

Oczywiście powyższe składowe są dostosowane do notacji graficznych i szczegółowych specyfikacji stosowanych w tej fazie.

Narzędzia CASE powinny automatyzować proces uszczegóławiania wyników analizy. Powinny np. umożliwiać automatyczne dodawanie pól/atrybutów realizujących związki klas/encji. Powinny także ułatwiać dostosowywanie projektu do ograniczeń środowiska implementacji. W przypadku implementacji z wykorzystaniem relacyjnych baz danych powinna np. istnieć możliwość automatycznej transformacji związków generalizacji-specjalizacji. Inaczej mówiąc, system CASE powinien zapewniać mechanizmy podobne do dziedziczenia w środowiskach obiektowych, wtedy gdy nie zapewnia tego środowisko implementacji.

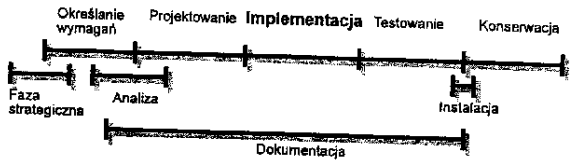
Niektóre narzędzia CASE zawierają narzędzia projektowania interfejsu użytkownika zbliżone do niezależnych narzędzi tego typu. Dzięki wykorzystywaniu informacji zawartych w słowniku danych możliwa jest pełniejsza automatyzacja prac nad interfejsem użytkownika. Narzędzia takie pozwalają na przykład automatycznie skonstruować dialog pozwalający na edycję pól danej klasy lub encji. Zadaniem projektanta pozostaje jedynie sformatowanie tego dialogu.

W fazie projektowania przydatne są także możliwości importu informacji z innych projektów, zawierających na przykład opisy bibliotek dostępnych w środowisku implementacji. Ułatwia to pełne wykorzystanie możliwości danego środowiska.

W fazie projektowania wykorzystywane są również narzędzia tzw. inżynierii odwrotnej. Pozwalają one na odtworzenie projektu, tj. specyfikacji i diagramów, na podstawie istniejącego kodu i struktury bazy danych. Mogą one np. zostać wykorzystane do automatycznej budowy projektu opisującego dostępne w danym środowisku biblioteki.

Rozdział 7. Implementacja

W fazie implementacji (kodowania) projekt oprogramowania jest realizowany w wybranym środowisku implementacji.



Rysunek 7.1. Faza implementacji w cyklu życia oprogramowania

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

- języków wysokiego poziomu,
- gotowych elementów,
- narzędzi szybkiego wytwarzania aplikacji — RAD,
- generatorów kodu.

Generatory kodu są składowymi narzędzi CASE, które na podstawie opisu projektu zawartego w słowniku danych automatycznie tworzą kod programu. Najczęściej jest to szkielet kodu, który musi być jeszcze uzupełniony przez programistów. Przykład automatycznie wygenerowanego kodu w C++ uzyskanego za pomocą pakietu Select OMT Professional Workbench firmy Select podany jest poniżej.

Przykład automatycznie wygenerowanego kodu dla języka C++

Plik nagłówkowy

```
#ifndef __PUNKT_H
#define __PUNKT_H

//{{SCG_HEADER(Punkt.h) [0]
////////////////////////////////////////////////////////////////////////
//
//
//  header file : Punkt.h
//
////////////////////////////////////////////////////////////////////////
//
#include "dib.h"
#include "symbol_.h"

class CPunktEditDlg;

//{{SCG_CLASS(0)
//{{SCG_CLASS_INFO(0)
////////////////////////////////////////////////////////////////////////
//
// Description:
//
// Constraints:
//

class CPunkt
                : public CSymbol_graficzny
//}}SCG_CLASS_INFO
{
    // Select generated class properties
    //{{SCG_CLASS_PROPS(0)
private:
    CSize Podstawowy_rozmiar;
public:
    CString Nazwa_pliku;
private:
    CPoint Polozenie;
    CString Rodzaj_symbolu;
public:
    void Wyswietl(CView* pView, CDC* pDC, const CRect
        Rzeczywiste_wspolrzedne, const CSize
        Wyswietlany_obszar, const BOOL Skalowanie,
        const long Powiekszenie);
    CPunkt(CMapa* Mapa);
    ~CPunkt();
    virtual BOOL Edytuj();
    void Umiesc();
    void Wybierz_plik();
    virtual void Zapis(CArchive& ar);
    void OK();
    CPunkt & operator=(const CPunkt& Punkt);
```



```
    virtual CRect Obszar_rysowania(CDC* pDC, const CRect
        Rzeczywiste_wspolrzedne, const CSize
        Wyswietlany_obszar, const BOOL Skalowanie,
        const long Powiekszenie);
    void Ustaw_polozenie(const CPoint Polozenie);
    virtual void Odczyt(CArchive& ar);
protected:
    CDIB DIB;
    CPunktEditDlg* PunktEditDlg;
    CPunkt* Chwilowy_punkt;
    CPunkt* Punkt;
    //}}SCG_CLASS_PROPS
};
//}}SCG_CLASS

//}}SCG_HEADER
#endif
```

Fragment pliku z implementacją

```
//{{SCG_IMPLEMENTATION(Punkt.cpp) [0]

////////////////////////////////////////////////////////////////////////
//
//
// implementation file : Punkt.cpp
//
////////////////////////////////////////////////////////////////////////
//
//{{SCG_INCLUDE
#include "punkt.h"
//}}SCG_INCLUDE

//{{SCG_OP(0.0)
//{{SCG_OP_INFO
////////////////////////////////////////////////////////////////////////
//
// Description:
// Wyświetl tło w podanym zakresie przewijalnego
// okna.
// Parameters_Description:
// pView - przewijalne okno, w którym wyświetlany jest symbol
// pDC - urządzenie,
// na którym symbol jest wyświetlany
// Rzeczywiste_wspolrzedne - rzeczywiste
// współrzędne wyświetlanej mapy, służą do obliczenia położenia
// symbolu w jednostkach Windows
// Wyswietlany_obszar - obszar w jakim wyświetlana jest
// mapa w jednostkach Windows
// bSkalowanie - czy symbol ma być skalowany
// w raz z mapą
// Powiekszenie - powiększenie wyświetlania

void CPunkt::Wyswietl(CView* pView, CDC* pDC, const CRect
    Rzeczywiste_wspolrzedne, const CSize Wyswietlany_obszar,
    const BOOL Skalowanie, const long Powiekszenie)
```

```
//}}SCG_OP_INFO
{
  // Code to be added here
}
//}}SCG_OP

//}}SCG_OPS

//}}SCG_IMPLEMENTATION
```

Faza implementacji w modelu kaskadowym polega nie tylko na zakodowaniu poszczególnych modułów, lecz także na ich przetestowaniu. Bezpośrednio po, a nawet w trakcie implementacji moduły są testowane przez programistów. Dalsze testy całego systemu wykonywane są w kolejnej fazie. Dla zachowania spójności książki wszystkie zagadnienia związane z testowaniem zostaną omówione w kolejnym rozdziale.

7.1. Programowanie niezawodnego oprogramowania — programowanie dla niezawodności

Faza implementacji ma duży wpływ na niezawodność oprogramowania. Dlatego w niniejszym rozdziale zostaną omówione sposoby zwiększania niezawodności oprogramowania. Znaczenie niezawodności wynika z:

- Rosnących oczekiwań klientów wynikających między innymi z faktu wysokiej niezawodności sprzętu. Użytkownicy przyzwyczajeni do wysokiej niezawodności komputerów zaczynają oczekiwać podobnej niezawodności od oprogramowania (nadal jednak, zwłaszcza w przypadku oprogramowania przeznaczonego dla komputerów osobistych, klienci akceptują znacznie niższy poziom niezawodności oprogramowania niż sprzętu).
- Potencjalnie bardzo dużych kosztów błędnych wykonań. Niepoprawna realizacja pewnych funkcji może być przyczyną wysokich strat finansowych, lub nawet zagrożenia dla życia ludzkiego.
- Nieprzewidywalności efektów oraz trudność usunięcia błędów w oprogramowaniu. W trakcie realizacji oprogramowania często pojawia się problem znalezienia kompromisu pomiędzy efektywnością i niezawodnością oprogramowania. Problemy związane z niewystarczającą efektywnością są jednak łatwo przewidywalne. Efektywność jest także stosunkowo łatwa do poprawy, np. poprzez optymalizację kluczowych fragmentów systemu, lub po prostu przez zmianę sprzętu na bardziej wydajny.

Zwiększenie niezawodności na etapie implementacji można osiągnąć dzięki: *unikaniu błędów* oraz *tolerancji błędów*.

7.1.1. Unikanie błędów

Pełne uniknięcie błędów na etapie implementacji wykonywanej przez programistów nie jest oczywiście możliwe. Pewne podejścia powalają jednak uniknąć wielu błędów. Podejścia takie to:

- *unikanie niebezpiecznych technik,*
- *zasada ograniczonego dostępu,*
- *stosowanie kompilatorów sprawdzających zgodność typów.*

Pewne techniki programistyczne zwiększają możliwość popełnienia błędów. Są to:

- *Instrukcja* **goto** lub inne o podobnym działaniu. Instrukcje te zaburzają naturalną strukturę algorytmu.
- *Liczby zmiennopozycyjne.* Obliczenia z wykorzystaniem liczb zmiennopozycyjnych wykonywane są ze skończoną dokładnością. Błędy zaokrągleń mogą być stosunkowo duże nawet w przypadku pojedynczych operacji, zwłaszcza wtedy, gdy poszczególne liczby zdecydowanie różnią się wielkością. Błędy zaokrągleń pojedynczych operacji mogą się kumulować prowadząc do bardzo poważnych błędów obliczeń. Wiele błędów wiąże się także z porównywaniem liczb zmiennopozycyjnych. Dwa wyrażenia, które z matematycznego punktu widzenia dają dokładnie ten sam wynik, mogą dać nieznacznie różniące się rezultaty w przypadku realizacji komputerowej.
- *Wskaźniki.* Posługiwanie się wskaźnikami prowadzi do wielu błędów dostępu do pamięci. Wskaźniki mogą prowadzić także do sytuacji, w których te same zmienne mają różne etykiety.
- *Obliczenia równoległe.* Stosowanie obliczeń równoległych prowadzi do złożonych zależności czasowych, które mocno utrudniają uruchamianie oprogramowania. Wiele błędów nie ujawnia się w momencie śledzenia programu. Możliwe jest także równoczesne operowanie na tych samych danych przez różne procesy.
- *Przerwania.* Technika ta wprowadza rodzaj równoległości, wiążą się więc z nią te same problemy jak z obliczeniami równoległymi. Dodatkowo istnieje ryzyko przerwania procesów krytycznych czasowo.
- *Rekursja.* Stosowanie tej techniki utrudnia śledzenie programu. Często jest także przyczyną zapętlania się programu.
- *Tryby pracy* procedur (funkcji, metod), które realizują wyraźnie odmienne zadania w zależności od informacji kontrolnej przekazanej jako parametry wejściowe.
- *Złożone wyrażenia wymagające uwzględnienia priorytetów operatorów.* Wielu błędów unika się rozbijając złożone wyrażenia na kilka krótszych lub zawsze ujmując podwyrażenia w nawiasy nawet wtedy, gdy (jak się programiście wydaje) nie jest to konieczne ze względu na priorytety operatorów.

Niektóre z powyższych technik mogą się w pewnych sytuacjach okazać bardzo przydatne lub nawet niezbędne. Nie można więc traktować ich jak technik zakazanych. Technik tych nie należy jednak nigdy stosować, jeżeli nie ma to istotnego uzasadnienia.

Wielu błędów można także uniknąć dzięki stosowaniu zasady ograniczonego dostępu. Zasada ta znana jest pod angielską nazwą *need-to-know* przejętą z armii amerykańskiej Zasada stosowana w armii oraz innych organizacjach dbających o bezpieczeństwo mówi, że wiedzieć (czyli mieć dostęp do informacji) powinny tylko te osoby, które muszą wiedzieć. Inaczej mówiąc, wszystkie informacje są z założenia utajnione chyba, że zachodzi wyraźna potrzeba udostępnienia ich komuś. Zasada ta przeniesiona na grunt programowania mówi, że dostęp do poszczególnych elementów programu powinny mieć tylko te inne składowe, które rzeczywiście odwołują się do tych elementów. Oznacza to np., że konkretne pole powinno być dostępne wyłącznie dla metod, które to pole odczytują lub ustawiają. Podobnie konkretna metoda powinna być dostępna wyłącznie dla metod, które ją wywołują. Typowe języki programowania nie pozwalają jednak na pełne stosowanie tej zasady. Należy jednak w miarę możliwości stosować wszelkie dostępne techniki ograniczania dostępu do składowych programu, np. poprzez stosowanie danych i funkcji prywatnych w modułach, stosowanie prywatnych pól i metod w klasach. Obiektowe języki programowania pozwalają na ograniczanie dostępu do pól i metod klasy, nawet jeżeli sama klasa jest dla danej składowej dostępna.

Przyczyną wielu błędów jest błędne deklarowanie typów zmiennych, parametrów oraz funkcji i omyłkowe wykonywanie operacji na wyrażeniach o różnych typach. Błędów tych można uniknąć, jeżeli kompilator w ścisły sposób sprawdza zgodność typów, czyli nie pozwala programiście na wykonanie błędnej operacji chyba, że konwersja typów zostanie jawnie wymuszona. Zasada uniemożliwiania użytkownikom wykonywania błędnych połączeń znana jest doskonale w innych dziedzinach inżynierii. Ilustruje to poniższy przykład:

> Tylna ściana komputera klasy PC zawiera szereg gniazd, do których można podłączyć szereg wtyczek. Konstruktorzy tych komputerów zadbali o to, aby każda wtyczka pasowała wyłącznie do jednego właściwego gniazdka (tak przynajmniej jest w komputerze, na którym powstaje ta książka). Jest oczywiste, że gdyby było inaczej — każdą wtyczkę można było włożyć do dowolnego gniazda — zdarzałyby się przypadki podłączenia zasilania do gniazda myszy i tym podobne sytuacje.

Kompilator sprawdzający zgodność typów powoduje więc, że zmienne oraz wyrażenia różnych typów „nie pasują" do siebie, podobnie jak wtyczka i gniazdo różnego rodzaju w powyższym przykładzie.

Konwersja typów bywa oczywiście niezbędna. Wielu programistów ma niestety tendencję do preferowania języków, które pozwalają na automatyczną konwersję typów. Jest to niewątpliwie jedna z przyczyn popularności języka C. Język C++ jest często nazywany językiem o ścisłej weryfikacji zgodności typów. W języku tym w pełni poprawny jest jednak następujący fragment programu:


```
typedef int TLiczba_krow;
typedef int TLiczba_jablek;
typedef int TLiczba_snopkow_siana;
...
TLiczba_krow Liczba_krow;
TLiczba_jablek Liczba_jablek;
TLiczba_snopkow_siana Liczba_snopkow_siana;
...
  Liczba_krow = Liczba_jablek + Liczba_snopkow_siana;
```

W języku C++, podobnie jak w innych językach obiektowych, klasa bazowa ma typ zgodny z klasami pochodnymi. Jest to oczywiście niezbędne z punktu widzenia pewnych mechanizmów programowania obiektowego, w pewnych sytuacjach może być jednak także przyczyną błędów.

Wiele kompilatorów zgodnie ze standardem danego języka programowania nie uznaje za błędy pewnych potencjalnie niebezpiecznych konstrukcji, między innymi naruszeń zgodności typów. Kompilatory te pozwalają jednak na generowanie ostrzeżeń w przypadku wykrycia takich konstrukcji. Z opcji tych należy korzystać w jak najszerszym zakresie. Wygenerowanych ostrzeżeń nie należy ignorować, nawet jeżeli ich pojawienie się nie przerywa kompilacji programu.

Chociaż język C++ został poddany powyżej pewnej krytyce, należy zaznaczyć, że języki obiektowe dostarczają możliwości tworzenia nowych typów, które nie będą zgodne z innymi typami. Język C++ np. pozwala na definiowanie operatorów operujących na danej klasie. Jeżeli pola klasy są prywatne, to posługiwanie się tymi operatorami jest jedynym sposobem dostępu do pól klasy.

7.1.2. Tolerancja błędów

Jest oczywiste, że żadna z powyższych technik unikania błędów nie gwarantuje uzyskania programu w pełni bezbłędnego. Tolerancja błędów oznacza, że program działa poprawnie, a przynajmniej sensownie także wtedy, gdy zawiera błędy. Tolerancja błędów wymaga wykonywania przez program następujących zadań:

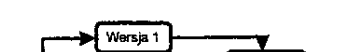
- wykrycia błędu,
- wyjścia z błędu, to jest zakończenia pracy modułu, w którym wystąpił błąd w poprawny sposób,
- ewentualnej naprawy błędu, to jest zmiany programu tak, aby zlikwidować wykryty błąd.

Istnieją dwa główne sposoby automatycznego wykrywania błędów:

- sprawdzanie warunków poprawności danych,
- porównywanie wyników różnych wersji modułu.

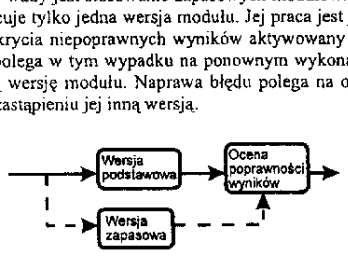
Jak wspomniano w sekcjach 5.3.4 i 5.5.4 omawiających sposób specyfikowania modeli obiektowych i strukturalnych, na etapie analizy i projektowania określa się między innymi ograniczenia jakie muszą spełniać poprawne dane (np. pola klasy) oraz warunki poprawnego rozpoczęcia i zakończenia metod i procesów. Naruszenie tych warunków oznacza wystąpienie błędu. Poprawność tych warunków może być sprawdzana przez dodatkowe fragmenty kodu. Pewne środowiska implementacji pozwalają na automatyczną weryfikację zdefiniowanych warunków poprawności. Wiele relacyjnych baz danych pozwala np. na sprawdzanie warunków integralności bazy danych.

Błędy można też wykryć wykorzystując kilka modułów wykonujących te same operacje. W elektronice znana jest zasada TMR (ang. triple-modular redundancy), stosowana w przypadku układów o szczególnie wysokich wymaganiach wobec niezawodności, np. w zastosowaniach kosmicznych. Zgodnie z tą zasadą wykorzystuje się trzy kopie (lub inną nieparzystą liczbę kopii) układu, którego poprawne działanie jest szczególnie istotne. Wyjścia tych trzech kopii są porównywane. Jeżeli nastąpi awaria jednej z nich, jest bardzo mało prawdopodobne, aby pojawiła się ona jednocześnie w jednej z pozostałych kopii. Dlatego też wynik, który różni się od dwóch pozostałych jest uznawany za błędny. Oprogramowanie nie ulega oczywiście awariom podobnym do awarii sprzętowych, tj. nie pojawiają się nim nowe błędy. Problemem są wyłącznie błędy istniejące w oprogramowaniu od momentu jego powstania. W związku z tym powielanie tego samego modułu w kilku kopiach nie miałoby oczywiście sensu. Poszczególne wersje danego modułu powinny zostać więc zaimplementowane przez niezależne zespoły programistów. Stosowanie wielu wersji modułów narzuca oczywisty sposób wyjścia z błędu poprzez wybór jako poprawnych wyników generowanych przez większość wersji. Technikę taką nazywa się programowaniem N-wersyjnym (porównaj rysunek 7.2).



Rysunek 7.2. Programowanie N-wersyjne

Większość systemów pracuje na komputerach sekwencyjnych. Wykonanie kilku wersji modułu wiąże się więc z dodatkowym narzutem czasu i zmniejsza wydajność systemu. Techniką pozbawioną tej wady jest stosowanie zapasowych modułów. W tym wypadku w danym momencie pracuje tylko jedna wersja modułu. Jej praca jest jednak monitorowana. W przypadku wykrycia niepoprawnych wyników aktywowany jest moduł zapasowy. Wyjście z błędu polega w tym wypadku na ponownym wykonaniu tych samych operacji przez zapasową wersję modułu. Naprawa błędu polega na odłączeniu wersji, w której wykryto błąd i zastąpieniu jej inną wersją.



Rysunek 7.3. Technika zapasowych modułów

Jeżeli nie wykorzystuje się wielu wersji modułów, podstawowym sposobem wyjścia z błędu jest powrót do poprzedniego poprawnego stanu. Większość systemów baz danych zapewnia akceptowanie wyników transakcji tylko wtedy, gdy wszystkie operacje zakończą się sukcesem. Powrót do poprzedniego stanu jest więc wykonywany automatycznie przez system zarządzania bazą danych. W innych środowiskach niezbędne może być przechowanie dodatkowych informacji, które umożliwią powrót do poprzedniego stanu.

7.2. Charakterystyka typowych środowisk implementacji

W rozdziale tym zawarta jest charakterystyka kilku podstawowych rodzajów środowisk implementacji. Oczywiście dany system może być implementowany z wykorzystaniem kilku środowisk różnego typu.

7.2.1. Języki proceduralne

Języki proceduralne są tradycyjnym i prawdopodobnie wciąż jeszcze najpopularniejszym środowiskiem implementacji. Nadają się szczególnie dobrze do implementacji projektu strukturalnego. Procesy i moduły wysokiego poziomu mogą odpowiadać całym aplikacjom. Procesy i moduły pośredniego poziomu — grupom procedur i funkcji. Procesy i moduły najniższego poziomu — pojedynczym procedurom i funkcjom. Zbiorniki danych odpowiadają strukturom danych języka proceduralnego lub plikom zewnętrznym.

Języki proceduralne z reguły dają niewielkie możliwości ograniczania dostępu do danych. W większości języków poszczególne składowe struktur danych są dostępne zawsze wtedy, gdy dostępna jest cała struktura. Na przykład w Pascalu nie ma możliwości zabezpieczania dostępu do pól rekordu. W językach tego typu z reguły można zabezpieczać dostęp do danych i procedur na poziomie modułów. Istnieją oczywiście języki proceduralne o znacznie bardziej rozbudowanych możliwościach ograniczania dostępu do procedur i danych — na przykład Ada.

Języki proceduralne pozwalają także na implementację projektu obiektowego. Prace nad metodami projektowania obiektowego rozpoczęto zresztą z myślą o Adzie (Booch, 1983), czyli języku proceduralnym. Pola klas umieszczane są wtedy w strukturach danych — na przykład rekordach. Metody klas implementowane są jako procedury i funkcje. Języki proceduralne nie zapewniają jednak wiązania danych z procedurami i funkcjami, które na nich operują. Implementacja projektu obiektowego w tego typu językach wymaga wzięcia pod uwagę wymienionych w sekcji 6.4 ograniczeń: braku dziedziczenia i braku metod wirtualnych.

7.2.2. Języki obiektowe

Języki obiektowe są oczywiście szczególnie przydatne jako środowisko implementacji projektu obiektowego. Z reguły pozwalają na realizację wszystkich elementów projektu obiektowego, chociaż pewne języki mogą nie zezwalać na wielokrotne dziedziczenie.

Kompilatory języków obiektowych często są wyposażone w biblioteki ułatwiające przechowywanie klas w plikach. Narzędzia te nie są jednak wystarczające, jeżeli niezbędne jest przechowywanie bardzo dużych zbiorów danych. W tym wypadku niezbędna staje się współpraca z systemem baz danych.

Większość języków obiektowych to języki hybrydowe, pozwalające na stosowanie technik proceduralnych. Języki tego typu nadają się więc równie dobrze do implementacji projektu strukturalnego jak typowe języki proceduralne. Zaletą stosowania hybrydowych języków obiektowych do realizacji projektu strukturalnego jest zwiększona możliwość ukrywania danych oraz możliwość definiowania nowych typów.

7.2.3. Relacyjne bazy danych

Relacyjne bazy danych to obecnie najlepiej rozwinięte środowiska pozwalające na przechowywanie dużych zbiorów informacji. Podstawowe zalety relacyjnych baz danych to: wielodostęp, automatyczna weryfikacja warunków poprawności, możliwość ograniczania dostępu poszczególnych użytkowników, wysoka niezawodność wiodących systemów, rozszerzalność, możliwość rozproszenia danych, możliwość usuwania kaskadowego powiązanych obiektów.

Wadami relacyjnych baz danych są:

- stosunkowo mała efektywność przy częstych odwołaniach do pojedynczych, wzajemnie powiązanych krotek z różnych relacji,
- brak typów złożonych i niewielki wybór typów podstawowych,
- niewielkie możliwości ograniczania dostępu do danych dla poszczególnych procedur,
- brak dziedziczenia.

Należy zaznaczyć, że wiele narzędzi CASE, zarówno strukturalnych, jak i obiektowych, w automatyczny sposób rozwiązujące problemy związane z brakiem typów złożonych i brakiem dziedziczenia.

Systemy zarządzania baz danych obejmują specjalizowane języki programowania, które z reguły są językami proceduralnymi. Nie są więc dobrym środowiskiem implementacji projektu obiektowego.

W ostatnim okresie rozwijane są standardy dostępu do relacyjnych baz danych. Przykładem jest standard ODBC (ang. *open database connectivity*). Wielu producentów baz danych wyposaża swoje produkty w biblioteki pozwalające na dostęp do danych z różnorodnych języków programowania. Pozwala to między innymi na dostęp do relacyjnych baz danych z poziomu języków obiektowych. Dzięki temu możliwa jest stosunkowo łatwa implementacja projektu obiektowego, połączona z przechowywaniem trwałych klas w relacyjnej bazie danych.

7.2.4. Obiektowe bazy danych

Z punktu widzenia programisty obiektowa baza danych powinna być typowym językiem obiektowym wyposażonym w możliwość automatycznego przechowywania trwałych klas oraz wyszukiwania obiektów na podstawie zapytań.

Dostępne obecnie obiektowe bazy danych nie osiągnęły jeszcze jakości porównywalnej z systemami relacyjnymi. Dlatego też trudno je zalecać jako środowiska implementacji. Z drugiej strony wiele relacyjnych baz danych wyposażanych jest w rozszerzenia „obiektowe", na przykład możliwość definiowania typów złożonych, wiązanie procedur z relacjami. Wydaje się więc, że to raczej relacyjne bazy danych będą ewoluowały w stronę środowisk obiektowych.

7.2.5. Środowiska programistyczne programów użytkowych

Programy użytkowe tradycyjnie były wyposażane w proste języki makrodefinicji, które pozwalały na lepsze dostosowanie programów do potrzeb konkretnego użytkownika. Obecnie jednak niektóre programy użytkowe stanowią w pełni rozbudowane środowiska programistyczne. Przykładem może być pakiet Microsoft Office. Program Microsoft Excel wchodzący w skład tego pakietu posiada np. następujące cechy:

- zawiera pełen proceduralny język programowania Visual Basic dla Aplikacji, pozwala także na wykorzystanie niezależnego języka Visual Basic,
- obejmuje bardzo rozbudowaną bibliotekę obiektów, udostępniającą praktycznie wszystkie możliwości tego programu,
- pozwala na nagrywanie makrodefinicji w formie procedur języka Visual Basic,
- posiada możliwości dialogowego projektowania interfejsu użytkownika, takie jak: projektowanie dialogów, menu i pasków narzędziowych, umieszczanie pól dialogowych na arkuszach, definiowanie reakcji na zajście rozmaitych zdarzeń,
- zawiera ułatwiający uruchamianie programów debugger,
- pozwala na dystrybucję aplikacji bez rozprowadzania kodu źródłowego,
- obejmuje rozbudowane możliwości współpracy z innymi programami, np. DLL, DDE, OLE, ODBC.

Środowiska programistyczne programów użytkowych zdecydowanie ułatwiają opracowanie prototypów, między innymi dlatego, że pozwalają szereg operacji zaprogramować w sposób czysto dialogowy przez nagranie makrodefinicji. Jeżeli tworzony system zawiera szereg funkcji dostępnych w danym programie użytkowym, to zastosowanie takiego środowiska może zdecydowanie zredukować czas realizacji systemu. Podejście takie można uznać za przykład montażu z gotowych elementów (porównaj sekcję 2.6 omawiającą ten model cyklu życia oprogramowania).

7.2.6. Narzędzia szybkiego wytwarzania aplikacji

Narzędzia szybkiego wytwarzania aplikacji — RAD pozwalają na tworzenie bezpośredniego połączenia pomiędzy elementami składowej kontaktu z użytkownikiem i elementami składowej zarządzania danymi. Pozwalają w dialogowy sposób projektować elementy interfejsu użytkownika: menu, dialogi, raporty. Poszczególne elementy interfejsu można przy tym wiązać z elementami składowej dziedziny problemu. Pola dialogów można np. łączyć z kolumnami w bazie danych. Narzędzia RAD pozwalają także na definiowanie podstawowych reakcji na akcje użytkownika, na przykład ukazanie się raportu po wybraniu opcji z menu. Wiele narzędzi tego typu obejmuje także języki programowania pomocne przy implementacji bardziej złożonych funkcji. Narzędzia te nadają się szczególnie dobrze do implementacji systemów nastawionych na przechowywanie i stosunkowo proste przetwarzanie danych.

7.3. Podsumowanie

7.3.1. Kluczowe czynniki sukcesu

Główne czynniki wpływające na sukces fazy implementacji to:

- *wysoka jakość i wystarczająca szczegółowość projektu,*
- *dobra znajomość środowiska implementacji,*
- *zachowanie przyjętych standardów,*
- *unikanie błędów.*

7.3.2. Podstawowe rezultaty fazy implementacji

Podstawowe rezultaty fazy projektowania to:

- poprawiony dokument opisujący wymagania,
- poprawiony model,

- poprawiony projekt, który od tej pory stanowi już dokumentację techniczną,
- kod składający się z przetestowanych modułów,
- raport opisujący testy modułów,
- dostrojona baza danych,
- harmonogram fazy testowania.

7.3.3. Narzędzia CASE w fazie implementacji

Programiści wykonujący implementację korzystają z dokumentacji projektu. Mogą przy tym zarówno korzystać z wydrukowanych dokumentów, jak i bezpośrednio przeglądać diagramy i słownik danych korzystając z narzędzia CASE.

Podstawowe składowe narzędzi CASE wykorzystywane w fazie implementacji to oczywiście generatory kodu, które na podstawie zawartości słownika danych generują szkielet kodu programu. Typowe elementy kodu, które mogą być generowane automatycznie to:

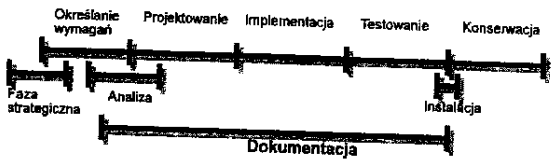
- skrypty tworzące relacje w bazie danych,
- definicje struktur danych,
- nagłówki procedur i funkcji,
- definicje klas,
- nagłówki metod.

Kod ten jest uzupełniany wieloma komentarzami tworzonymi na podstawie informacji zawartych w słowniku danych.

Niektóre narzędzia CASE posiadają sprzęgi do narzędzi RAD, co zapewnia dodatkowe przyspieszenie procesu implementacji.

Rozdział 8. Dokumentacja

W omawianej w tym rozdziale fazie tworzona jest dokumentacja użytkownika, którą należy traktować jako integralną składową końcowego produktu. Faza ta jest wykonywana równolegle z produkcją oprogramowania. Może się ona rozpocząć już w trakcie określania wymagań. Jak wspomniano w sekcji 2.1 omawiającej model kaskadowy cyklu życia oprogramowania, sugeruje się nawet, że podręcznik użytkownika dla przyszłego systemu jest dobrym dokumentem opisującym wymagania. Ostatnie zmiany w dokumentacji dokonywane są w fazie instalacji.



Rysunek 8.1. Faza dokumentacji w cyklu życia oprogramowania

8.1. Składowe dokumentacji użytkowej

Dokumentacja użytkowa jest przeznaczona dla różnych klas odbiorców. Dwie podstawowe klasy to

- użytkownicy końcowi,
- administratorzy systemu.

Użytkownicy końcowi mogą różnić się stopniem zaawansowania. Mogą też wykorzystywać system w różny sposób i w związku z tym poszukiwać w dokumentacji różnych informacji. Można więc wyróżnić dalsze klasy użytkowników końcowych.

Podstawowe składowe dokumentacji użytkowej to:

- *Opis funkcjonalny.* Jest to wstępna część dokumentacji, która w zwarty sposób opisuje przeznaczenie i główne możliwości systemu. Opis funkcjonalny powinien dostarczyć osobie rozważającej zakup lub wykorzystanie systemu niezbędnych informacji pozwalających ocenić czy system spełnia jej potrzeby. Opis funkcjonalny jest też przydatny początkującym użytkownikom, którzy nie znają jeszcze dobrze możliwości systemu.
- *Podręcznik użytkownika.* Jest to opis systemu przeznaczony głównie dla początkujących użytkowników. Część ta powinna zawierać informacje o:
  - sposobach uruchamiania oraz kończenia pracy z systemem,
  - sposobach realizacji najczęściej wykorzystywanych funkcji systemu,
  - metodach obsługi błędów, np. o sposobach odwoływania błędnych operacji wykonanych przez użytkownika,
  - sposobach korzystania z systemu pomocy.

Podręcznik użytkownika powinien przedstawiać prosty przykład korzystania z systemu.

- *Kompletny opis.* Jest to część przeznaczona głównie dla doświadczonych użytkowników. Powinna zawierać
  - szczegółowy opis wszystkich funkcji systemu,
  - informacje o wszystkich sposobach wywoływania tych funkcji,
  - opisy formatów danych,
  - opisy błędów, które mogą się pojawiać podczas pracy z systemem,
  - informacje o wszelkich ograniczeniach dotyczących np. zakresów danych.
- *Opis instalacji.* Jest to składowa dokumentacji przeznaczona głównie dla administratora systemu. Powinna zawierać opis procedury instalacji oraz dostrojenia systemu do środowiska, w którym system będzie pracować.
- *Podręcznik administratora systemu.* Część ta powinna opisywać możliwości zmian konfiguracji systemu i sposoby udostępniania systemu użytkownikom końcowym.

W przypadku dużych systemów złożonych z wielu odrębnych programów, może oczywiście powstać wiele podręczników opisujących poszczególne programy.

Zarówno całość dokumentacji, jak i poszczególne podręczniki mogą dodatkowo zawierać:

- słownik używanych terminów,
- indeks.

Dokumentacja użytkowa obejmuje także opisy systemu dostępne w formie elektronicznej w trakcie korzystania z systemu. Dokumentacja elektroniczna najczęściej jest budowana na podstawie kompletnego opisu systemu. Jest więc przeznaczona przede wszystkim dla doświadczonych użytkowników, którzy chcą szybko odnaleźć informacje na interesujący ich temat. Możliwości hipertekstu pozwalają na umieszczanie na poszczególnych stronach odwołań do powiązanych opisów, dzięki czemu wyszukanie potrzebnej informacji może być znacznie szybsze niż w przypadku korzystania z dokumentacji w formie drukowanej.

Obecnie coraz częściej spotyka się także dokumentację elektroniczną przeznaczoną dla początkujących użytkowników zawierającą podstawowe informacje o systemie. Przybiera ona często formę specjalnych programów — przewodników, które asystują użytkownikowi w pracy nad przykładowym problemem podpowiadając mu dalsze kroki postępowania. Przykładem takiego narzędzia jest moduł Cue Cards pakietu Microsoft Project.

8.2. Jakość dokumentacji

Na jakość dokumentacji wpływają następujące czynniki:

- *Struktura.* Poszczególne podręczniki powinny być w czytelny i zrozumiały sposób podzielone na rozdziały, podrozdziały i sekcje.
- *Zachowanie standardów.* Poszczególne podręczniki oraz fragmenty tego samego podręcznika powinny mieć spójną strukturę, formę i sposób pisania.
- *Sposób pisania.*

Na jakość sposobu pisania wpływają następujące elementy:

- *Stosowanie formy aktywnej oraz zwracanie się bezpośrednio do czytelnika.* Wynika z tego, że zamiast zdań:

  Na ekranie pojawi się dialog ...

  Aby otworzyć plik należy wybrać z menu...

  powinny się pojawić zdania:

  Na ekranie zobaczysz dialog...

  Jeżeli chcesz otworzyć plik, wybierz z menu...

- *Poprawność gramatyczna i ortograficzna.* Wszelkie błędy w dokumentacji obniżają zaufanie użytkownika do całego systemu. Obecnie wiele edytorów tekstu posiada narzędzia sprawdzania poprawności językowej (głównie ortografii), które pozwalają wykryć wiele błędów.
- *Krótkie zdania.* Nie zaleca się stosowania w dokumentacji użytkowej długich, złożonych zdań, które są zdecydowanie mniej czytelne niż kilka krótkich zdań zawierających pojedyncze fakty.

- *Krótkie akapity.* Najlepiej zauważalne są informacje umieszczone na początku i na końcu akapitu. Im dalej od początku i końca akapitu umieszczona jest dana informacja, tym mniejsza szansa, że zostanie ona zauważona przez czytelnika. Informacje zawarte wewnątrz długich akapitów mają więc najmniejsze szanse na zauważenie. Wynika z tego, że akapity powinny być krótkie. Najlepiej jest, jeżeli pierwsze i ostatnie zdania akapitu stanowią swego rodzaju wprowadzenie i podsumowanie informacji zawartych w akapicie.

- *Oszczędność słów.* Tekst dokumentacji użytkowej powinien być przede wszystkim przejrzysty i precyzyjny. Poszczególne fakty należy wyrażać więc w jak najkrótszy sposób. W tekstach literackich zaleca się unikanie częstych powtórzeń tych samych słów i zwrotów. Powtórzeń tych można uniknąć stosując np. synonimy. Zastępowanie słów i zwrotów synonimami może jednak obniżać precyzję i przejrzystość tekstu. Powtórzenia należy więc uznać za dopuszczalne w dokumentacji użytkowej, jeżeli dla danego terminu nie można znaleźć jednoznacznego odpowiednika.

- *Precyzyjna definicja używanych terminów.* W dokumentacji może pojawić się wiele terminów, które nie są powszechnie znane. Pewne terminy mogą z kolei być używane w ramach dokumentacji w tylko jednym z wielu znaczeń w jakich funkcjonują w języku codziennym. Terminy tego typu powinny być precyzyjnie definiowane, a ich definicje powinny być zebrane w słowniku.

- *Powtarzanie trudnego opisu.* Szczególnie ważne informacje, których opis może być trudny dla czytelnika można powtarzać w różnych miejscach dokumentacji.

- *Stosowanie tytułów i podtytułów sekcji, wyliczeń i wyróżnień.* Wymienione elementy ożywiają tekst i pozwalają zwrócić uwagę czytelnika na najistotniejsze informacje.

- *Zrozumiałe odwołania do innych rozdziałów.* Odwołując się do rozdziału należy podać nie tylko jego numer, lecz także krótki opis zawartości. Odwołanie nie powinno więc wyglądać w następujący sposób:

  W rozdziale 5.1. znajduje się ...

  lecz następująco:

  W rozdziale 5.1. omawiającym zagadnienia ... znajduje się...

Komentarza wymaga podane jako pierwsze zalecenie zwracania się bezpośrednio do czytelnika. Jest to zasada stworzona z myślą o języku angielskim, który ma zupełnie inne tradycje niż język polski. Stosowanie tej zasady może wywołać obiekcje u części czytelników. Wydaje się, że rozsądnym kompromisem jest przestrzeganie tej zasady w podręczniku użytkownika. W kompletnym opisie lepsza wydaje się forma bezosobowa.

8.3. Podsumowanie

8.3.1. Kluczowe czynniki sukcesu

Najistotniejsze elementy wpływające na sukces fazy dokumentacji to:

- *wysoka jakość definicji wymagań, modelu i projektu,*
- *uwzględnienie różnego stopnia zaawansowania użytkowników.*

8.3.2. Podstawowe rezultaty fazy dokumentacji

Wynikiem fazy dokumentacji jest dokumentacja użytkowa w formie drukowanej i elektronicznej.

W trakcie prac nad dokumentacją użytkową często wykrywane są też błędy i nieścisłości w definicji wymagań, modelu i projekcie.

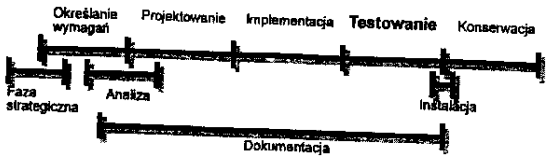
8.3.3. Narzędzia CASE w fazie dokumentacji

Autorzy dokumentacji użytkowej korzystają z definicji wymagań, modelu i projektu budowanych przy pomocy narzędzi CASE. Wykorzystują więc możliwości przeglądania słownika danych, generatory raportów i narzędzia przygotowywania dokumentacji technicznej.

Rozdział 9. Testowanie

Mówiąc o testowaniu należy rozróżnić:

- *atestowanie* (ang. *validation*), czyli testowanie zgodności systemu z rzeczywistymi potrzebami użytkownika,
- *weryfikację* (ang. *verification*), czyli testowanie zgodności systemu z wymaganiami zdefiniowanymi w fazie określania wymagań.



Rysunek 9.1. Faza testowania w cyklu życia oprogramowania

Testowanie ma dwa główne cele:

- wykrycie i usunięcie błędów w systemie,
- ocenę niezawodności systemu.

Niezbędne jest wyraźne rozróżnienie dwóch terminów: błędu (ang. *fault*, *error*) i błędnego wykonania (ang. *failure*).

Błąd

Błąd jest niepoprawną konstrukcją znajdującą się w programie, mogącą prowadzić do niewłaściwego działania.

Błędne wykonanie

Błędne wykonanie to niepoprawne działanie systemu w trakcie jego pracy.

Błąd może prowadzić do wielu różnych błędnych wykonań. W wielu sytuacjach może jednak nie objawiać się błędnym wykonaniem. Takie same błędne wykonania mogą być spowodowane różnymi błędami.

Testy można klasyfikować z różnych punktów widzenia. Z punktu widzenia głównego celu testy można podzielić na:

- *wykrywanie błędów*, czyli testy, których głównym celem jest wykrycie jak największej liczby błędów w programie,
- *testy statystyczne*, których celem jest wykrycie przyczyn najczęstszych błędnych wykonań oraz ocena niezawodności systemu.

Z punktu widzenia podstawowej techniki wykonywania testów można je podzielić na:

- *testy dynamiczne*, które polegają na wykonywaniu (fragmentu) programu i porównywaniu uzyskanych wyników z wynikami poprawnymi,
- *testy statyczne*, oparte na analizie kodu.

Testy dynamiczne pozwalają oczywiście wykryć wyłącznie błędne wykonania. Ustalenie ich przyczyny może wymagać wielu dalszych testów i/lub statycznej analizy kodu.

Testowanie systemu przebiega z reguły przez następujące fazy:

- *Testy modułów*. Są one wykonywane już w fazie implementacji bezpośrednio po zakończeniu realizacji poszczególnych modułów.
- *Testy systemu*. W fazie tej integrowane są poszczególny moduły i testowane są poszczególne podsystemy oraz system jako całość.
- *Testy akceptacji* (ang. *acceptance testing*). W przypadku oprogramowania realizowanego na zamówienie system przekazywany jest do przetestowania przyszłemu użytkownikowi. Testy takie nazywa się wtedy testami alfa. W przypadku oprogramowania sprzedawanego rynkowo testy takie polegają na nieodpłatnym przekazaniu pewnej liczby kopii systemu grupie użytkowników. Testy takie nazywa się testami beta.

9.1. Testy statystyczne

Testy statystyczne przebiegają według następującego schematu:

- losowa konstrukcja danych wejściowych zgodnie z rozkładem prawdopodobieństwa tych danych,
- określenie wyników poprawnego działania systemu na tych danych,
- uruchomienie systemu oraz porównanie wyników jego działania z poprawnymi wynikami.

Powyższe czynności wykonywane są cyklicznie.

Stosowanie testów statystycznych wymaga określenia rozkładu prawdopodobieństwa danych wejściowych możliwie bliskiego rozkładowi, który pojawi się podczas rzeczywistego korzystania z systemu. Dokładne określenie tego rozkładu jest bardzo trudne, często praktycznie niemożliwe. Wnioski dotyczące niezawodności oprogramowania wyciągnięte na podstawie analizy wyników testów mogą okazać się chybione, jeżeli rozkład prawdopodobieństwa generowanych danych wejściowych mocno odbiega od rozkładu danych rzeczywistych.

Założeniem testów statystycznych jest położenie nacisku na przetestowanie działania systemu w typowych sytuacjach. Wynika to z tego, że typowe dane wejściowe generowane są częściej niż nietypowe. Jeżeli jednak zachowanie systemu w pewnych sytuacjach jest szczególnie ważne, można oczywiście zmienić rozkład prawdopodobieństwa tak, aby osiągnąć częstsze testowanie takich sytuacji.

Główną zaletą testów statystycznych jest łatwość ich automatyzacji. Jeśli zostanie ustalony rozkład prawdopodobieństwa danych wejściowych oraz sposób określania pożądanego działania systemu, to dalsze testowanie może przebiegać automatycznie. Testy nastawione na wykrywanie błędów są z reguły wykonywane przy ścisłym udziale ludzi. Testy statystyczne można więc uznać za przykład techniki „brutalnej siły". Technika ta jest stosunkowo mało efektywna. Jednak dzięki zautomatyzowaniu tych testów można wykonać bardzo dużą liczbę przebiegów testowych.

Obserwacje liczby błędnych wykonań podczas testów statystycznych pozwalają określić niezawodność oprogramowania. Jako miary niezawodności wykorzystuje się:

- *Prawdopodobieństwo błędnego wykonania podczas realizacji transakcji*. W przypadku pewnych systemów wykonywane operacje mają charakter transakcji, które powinny zakończyć się pełnym sukcesem lub zostać zaniechane. Każde błędne wykonanie należy uznać za niepowodzenie całej transakcji. Miarę tę można oszacować na podstawie częstości występowania transakcji, które nie powiodły się z powodu jednego lub kilku błędów.
- *Częstotliwość występowania błędnych wykonań*. Miara ta określa spodziewaną liczbę błędnych wykonań w jednostce czasu. Przykładowo wielkość tej miary 0.1/h oznacza, że w ciągu godziny spodziewana liczba błędnych wykonań wynosi 0.1. Miara ta jest najczęściej stosowana w przypadku systemów, które nie mają charakteru transakcyjnego. Szacując tę miarę na podstawie testów należy wziąć pod uwagę fakt różną intensywność użytkowania systemu w czasie testowania i podczas praktycznego użytkowania. Dotyczy to szczególnie testów wykonywanych w pełni automatycznie. W trakcie testów polecenia mogą być wydawane a dane wprowadzane, w tempie zdecydowanie większym niż podczas rzeczywistego użytkowania.
- *Średni czas między błędnymi wykonaniami*. Miara jest odwrotnością poprzedniej.
- *Dostępność*. Miara ta opisuje prawdopodobieństwo, że w danej chwili system będzie dostępny do użytkowania. Miarę tę można oszacować na podstawie stosunku czasu, w którym system jest dostępny do czasu przeprowadzania testów. Na

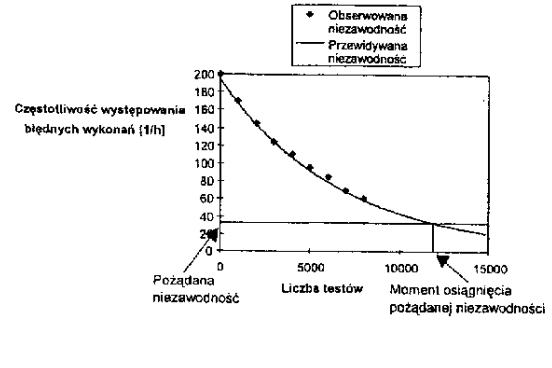
miarę tę wpływają wyłącznie te błędne wykonania, które prowadzą do czasowej niedostępności systemu. Wielkość tej miary zależy nie tylko od liczby błędnych wykonań, lecz także od szybkości powrotu do stanu normalnego po zajściu błędnego wykonania.

Oszacowanie niezawodności systemu pozwala między innymi określić czy osiągnięty został już wymagany przez klienta poziom niezawodności. Informacja o niezawodności systemu jest przydatna także wtedy, gdy klient nie określił wymagań wobec niezawodności. Częstotliwość występowania błędnych wykonań ma duży wpływ na koszty konserwacji oprogramowania. Wiele firm sprzedających oprogramowanie rynkowo oferuje swoim klientom serwis telefoniczny. Znajomość niezawodności oprogramowania oraz liczby potencjalnych nabywców pozwala oszacować liczbę zgłoszeń telefonicznych, które firma otrzyma. Na tej podstawie można oszacować niezbędną liczbę pracowników w dziale serwisu telefonicznego, a następnie koszty ich pracy.

Po wykryciu błędnych wykonań poszukiwane są ich przyczyny, to jest błędy, które są następnie usuwane z programu. Jeżeli nie wprowadza się przy tym równej lub większej liczby błędów, to w trakcie testowania wzrasta niezawodność oprogramowania. Jeżeli wykonywane są testy czysto statystyczne, wzrost niezawodności powinien przebiegać zgodnie z następującym modelem:

Niezawodność = Niezawodność początkowa × exp (-C × Liczba testów).

Jest to tzw. logarytmiczny model wzrostu niezawodności oprogramowania. W powyższym wyrażeniu zakłada się, że miarą niezawodności jest częstotliwość występowania błędnych wykonań. Stała C zależy do konkretnego systemu. Można ją określić na podstawie obserwacji niezawodności systemu w trakcie procesu testowania metodą regresji nieliniowej, np. stosując technikę najmniejszych kwadratów. Wygodnym narzędziem służącym do tego celu są typowe arkusze kalkulacyjne i programy statystyczne. Ekstrapolując w przyszłość obserwowany wzrost niezawodności można oszacować czas osiągnięcia pożądanej niezawodności, tj. niezawodności wymaganej przez klienta lub niezawodności pożądanej ze względu analizę kosztów konserwacji.



Rysunek 9.2. Logarytmiczny model wzrostu niezawodności oprogramowania

Szybszy wzrost niezawodności można osiągnąć, jeżeli dane testowe nie są dobierane w pełni losowo, lecz w kolejnych przebiegach testuje się działanie systemu w sytuacjach, które nie były jeszcze przetestowane. W idealnym przypadku osiągany jest wtedy wzrost niezawodności zgodny z modelem liniowym:

Niezawodność = Niezawodność początkowa – D × Liczba testów.

Osiągnięcie tak szybkiego wzrostu niezawodności nie jest możliwe w większości praktycznych sytuacji. Techniki testowania, które starają się zapewnić wzrost niezawodności szybszy od logarytmicznego omawiane są w kolejnym rozdziale.

9.2. Wykrywanie błędów

Dynamiczne testy zorientowane na wykrywanie błędów dzieli się na:

- *Testy funkcjonalne* (ang. *functional tests*, *black-box tests*), które zakładają znajomość jedynie wymagań wobec testowanych funkcji. System traktowany jest więc jako czarna skrzynka, która w nieznany sposób realizuje wymagane funkcje. Testy tego typu muszą być wykonywane przez osoby, które nie były zaangażowane w realizację testowanych fragmentów systemu.
- *Testy strukturalne* (ang. *structural tests*, *white-box tests*, *glass-box tests*), które zakładają znajomość sposobu implementacji testowanych funkcji.

9.2.1. Testy funkcjonalne

Pełne przetestowanie rzeczywistego systemu praktycznie nigdy nie jest możliwe z powodu olbrzymiej liczby dopuszczalnych danych wejściowych. U podstaw testów funkcjonalnych leży obserwacja, że dane wejściowe można podzielić na pewne klasy, dla których działanie systemu powinno być podobne. Jeżeli testowana funkcja działa poprawnie dla kilku danych należących do takiej klasy, zakłada się, że działanie systemu jest poprawne dla całej klasy danych wejściowych. Oczywiście jest to wnioskowanie czysto heurystyczne. Fakt poprawnego działania w kilku przebiegach nie gwarantuje, że błędne wykonania nie pojawią się dla innych danych z tej samej klasy.

Podział danych wejściowych na klasy odbywa się na podstawie analizy opisu wymagań. Następujący opis:

> Rachunek o wartości do 1000 zł może być zatwierdzony przez kierownika niższego szczebla. Rachunki o wartości powyżej tysiąca złotych muszą być akceptowane przez kierownika wyższego szczebla.

sugeruje podział danych wejściowych na dwie klasy w zależności od wartości rachunku. Testując odpowiednią funkcję należałoby więc wprowadzić informację o co najmniej jednym rachunku o wartości do tysiąca złotych i co najmniej jednym o wartości powyżej tej sumy. Wykonanie w tym wypadku tylko dwóch testów nie jest jednak zalecane.

Niezbędne jest przetestowanie funkcji dla *granicznych wartości* danych. W omawianym przykładzie należałoby więc wprowadzić także informacje o rachunku, którego wartość wyniesie dokładnie 1000 zł. Wykonywanie testów dla wartości granicznych jest podstawową zasadą zarówno testów funkcjonalnych, jak i strukturalnych.

Dzieląc dane wejściowe na klasy należy brać pod uwagę rozmaite kombinacje elementarnych warunków. Jeżeli wymienione powyżej wymaganie dotyczące rachunków uzupełnione jest następującym opisem:

> Kierownik niższego szczebla może w ciągu miesiąca zaakceptować rachunki o łącznej wartości do 10000 zł. Każdy rachunek powodujący przekroczenie tej sumy musi być zaakceptowany przez kierownika wyższego szczebla.

to wśród danych wejściowych można wyróżnić następujące klasy:

- rachunek o wartości do 1000 zł, nie powodujący przekroczenia miesięcznego limitu,
- rachunek o wartości do 1000 zł, powodujący przekroczenie miesięcznego limitu,
- rachunek o wartości powyżej 1000 zł, nie powodujący przekroczenia miesięcznego limitu,
- rachunek o wartości powyżej 1000 zł, powodujący przekroczenie miesięcznego limitu.

Jeżeli dodatkowo uwzględni się wartości graniczne, to niezbędne staje się wykonanie 9 testów. Kolejne warunki mogą spowodować dalszy gwałtowny wzrost liczby klas danych wejściowych. W praktyce więc przetestowanie wszystkich możliwych kombinacji może okazać się niemożliwe Dlatego też niezbędny staje się wybór testowanych funkcji i danych wejściowych. Ogólne warunki takiego wyboru są następujące:

- *Możliwość wykonania funkcji jest ważniejsza niż jakość jej wykonania.* Brak możliwości wykonania pewnej funkcji jest poważniejszym błędem, niż na przykład niepoprawne odświeżanie ekranu podczas jej realizacji.
- *Funkcje systemu znajdujące się w poprzedniej wersji są istotniejsze niż nowo wprowadzone.* Użytkownicy, którzy korzystali z poprzedniej wersji będą szczególnie niezadowoleni, jeżeli posługując się nową wersją nie będą mogli wykonać operacji, które wykonywali poprzednio.
- *Typowe sytuacje są ważniejsze niż wyjątki.* Błąd w funkcji wykonywanej przez typowego użytkownika co kilka minut jest więc ważniejszy niż błąd w funkcji wykonywanej raz w miesiącu.

9.2.2. Testy strukturalne

W przypadku testów strukturalnych, dane wejściowe dobiera się na podstawie analizy struktury programu realizującego testowane funkcje. Proponuje się w tym wypadku kilka kryteriów doboru danych testowych.

Kryterium pokrycia wszystkich instrukcji

Zgodnie z tym kryterium dane wejściowe należy dobierać tak, aby każda instrukcja została wykonana co najmniej raz. Spełnienie tego kryterium z reguły wymaga wykonania niewielkiej liczby testów. O tym, że spełniając to kryterium łatwo jest pominąć pewne błędy świadczy podany poniżej przykład:

```
if x > 0 then
begin
    ...
end
y = ln (x); {Logarytm naturalny}
```

Testując powyższą fragment programu wystarczy podać takie dane, aby zmienna x miała wartość większą od zera. Zapewnia to wykonanie wszystkich instrukcji. Nie zapewnia jednak wykrycia błędu arytmetycznego, który pojawia się, jeżeli wartość tej zmiennej jest równa lub mniejsza od zera.

Kryterium pokrycia instrukcji warunkowych

Kryterium to mówi, że dane wejściowe należy dobierać tak, aby każdy elementarny warunek instrukcji warunkowej został co najmniej raz spełniony i co najmniej raz nie spełniony. Test należy wykonać także dla wartości granicznej każdego warunku.

Zastosowanie tego kryterium pozwoliłoby wykryć błąd z poprzedniego przykładu, gdyż konieczne stałoby się podanie parametru o wartości zero lub mniejszej od zera. Proponuje się oczywiście także bardziej zaawansowane kryteria prowadzące do bardziej wyczerpujących testów, takie jak: kryterium pokrycia warunków elementarnych, kryterium pokrycia wszystkich niezależnych ścieżek w grafie przepływów programu czy kryterium pokrycia wszystkich ścieżek (Sommerville, 1995; McCabe, 1983). Poniższy przykład zawiera jednak błąd, który jest bardzo trudny do wykrycia przy zastosowaniu technik testowania strukturalnego. Poniższy program powinien znaleźć wszystkie rozwiązania rzeczywiste równania $ax^2+bx+c=0$, dla dowolnych wartości a, b, i c.

```
Program rownanie;
var
    a, b, c, delta : real;
begin
    readln (a, b, c);
    delta := b*b - 4 * a * c;
    if delta < 0 then
      writeln ('Brak pierwiastków rzeczywistych')
    else
    if delta = 0 then
        writeln ('X0 = ', -b / (2 * a))
    else
        writeln ('X1 = ', (-b - sqrt (delta)) / (2 * a),
        'X2 = ', (-b - sqrt (delta)) / (2 * a))
end.
```

Program nie działa poprawnie, gdy parametr *a* ma wartość 0. Przykład ten wykazuje istotną wadę testów strukturalnych. Jeżeli autor programu popełnił błąd polegający na nie uwzględnieniu istnienia pewnej klasy danych wejściowych (w tym wypadku klasy danych, dla których *a*=0), to błąd taki ma niewielkie szanse wykrycia.

Szczególne reguły proponuje się w przypadku testowania programów zawierających pętle. W tym wypadku zaleca się dobranie danych wejściowych tak, aby:

- nie została wykonana żadna iteracja pętli lub, jeżeli nie jest to możliwe, została wykonana minimalna liczba iteracji,
- została wykonana maksymalna liczba iteracji,
- została wykonana przeciętna liczba iteracji.

9.2.3. Testy statyczne

Testy statyczne polegają na analizie kodu bez uruchamiania programu. Stosowane techniki to:

- *dowody poprawności*,
- *metody nieformalne*.

Stosowanie formalnych dowodów poprawności jest techniką sugerowaną już od wielu lat (Dijkstra, 1976, McGettrick, 1982). Jest to jednak technika bardzo złożona. Ocenia się, że udowodnienie poprawności fragmentu programu jest o rząd trudniejsze i bardziej czasochłonne niż jego opracowanie. Wynika z tego nie tylko duży koszt stosowania tej techniki lecz również ryzyko popełnienia błędów na etapie tworzenia dowodu. Obecnie jest to technika bardzo rzadko stosowana w praktyce (Sommerville, 1995).

Statyczne testy nieformalne polegają na analizie kodu przez programistów. Proponuje się dwa niewykluczające się podejścia:

- *śledzenie przebiegu programu*,
- *wyszukiwanie typowych błędów*.

Pierwsze z tych podejść polega na tym, że osoba lub grupa osób analizująca kod stara się prześledzić przebieg działania programu zwracając przy tym uwagę na ewentualne błędy. Inaczej mówiąc program jest uruchamiany „w umyśle" testującej osoby (osób).

Wyszukiwanie błędów polega na przeglądaniu kodu bez śledzenia jego działania. Zwraca się przy tym uwagę na typowe błędy popełniane przez programistów. Przed przystąpieniem do analizy kodu warto przygotować sobie listę typowych błędów, na przykład:

- niezainicjowane zmienne,
- porównania liczb zmiennopozycyjnych,

- indeksy wykraczające poza tablice,
- błędne operacje na wskaźnikach,
- błędy w warunkach instrukcji warunkowych,
- niekończące się pętle,
- błędy popełnione dla granicznych wartości (na przykład >= zamiast >),
- błędne użycie lub pominięcie nawiasów w złożonych wyrażeniach,
- nieuwzględnienie błędnych danych.

Testy nieformalne mogą być wykonywane zarówno przez pojedyncze osoby, jak i przez grupy osób podczas specjalnych spotkań. Jedna z możliwych strategii stosowania testów nieformalnych jest następująca:

- Programista, który dokonał implementacji danego modułu w nieformalny sposób analizuje jego kod.
- Kod uznany przez autora implementacji za nie zawierający zbyt wielu błędów jest analizowany przez doświadczonego programistę, np. szefa zespołu programistów. Jeżeli już na wstępie okaże się, ze moduł zawiera wiele błędów jest on zwracany do przejrzenia autorowi.
- Szczególnie istotne moduły są analizowane przez grupę osób.

Techniki nieformalne okazują się być bardzo efektywne w praktyce. Praktyczne eksperymenty porównawcze różnych technik wykrywania błędów (Fagan, 1986) wskazują, że to właśnie techniki nieformalne pozwalają wykryć największą liczbę błędów w najkrótszym czasie. Moje doświadczenia potwierdzają tę obserwację. Jednocześnie wydaje się, że techniki nieformalne są niedoceniane w praktyce i zbyt rzadko używane podczas testowania.

Wymienione powyżej eksperymenty porównawcze wskazują także, że testy funkcjonalne są efektywniejsze niż strukturalne. Na pierwszy rzut oka wynik ten może okazać się zaskakujący. Testy strukturalne opierają się przecież na bogatszej informacji niż testy funkcjonalne. Okazuje się jednak, że błędy „zaszyte" w strukturze programu są bardzo trudne do wykrycia przy wykorzystaniu technik strukturalnych.

9.3. Ocena liczby błędów

Liczba błędów w programie nie musi mieć jednoznacznego związku z niezawodnością oprogramowania. Z punktu widzenia użytkownika ocena liczby błędów wydaje się więc nie mieć dużego znaczenia. Oszacowanie liczby błędów może mieć jednak spore znaczenia dla producenta oprogramowania. Liczba błędów ma bezpośredni wpływ na koszty konserwacji oprogramowania.

Znając:

- szacunkową liczbę błędów w programie,
- średni procent błędów zgłaszanych przez użytkowników systemu, oszacowany na podstawie danych z poprzednich przedsięwzięć,
- średni koszt usunięcia błędu także pochodzący z analizy danych z poprzednich przedsięwzięć,

można oszacować koszt konserwacji związany z usuwaniem błędów. Jest to szczególnie istotne dla firm sprzedających oprogramowanie pojedynczym lub nielicznym użytkownikom W tym wypadku koszt związany z pojawianiem się informacji o błędnych wykonaniach (np. koszt serwisu telefonicznego) jest niewielki w porównaniu z kosztami usuwania błędów.

Na oszacowanie liczby błędów w programie pozwala technika posiewania błędów. Polega ona na wprowadzeniu do programu pewnej liczby błędów podobnych do błędów, które występują w programie. Następnie wykonywana jest seria testów przeprowadzanych oczywiście przez osoby, które nie brały udziału w posiewaniu błędów. Na podstawie liczby wszystkich znalezionych błędów oraz liczby znalezionych błędów sztucznie wprowadzonych, można oszacować liczbę błędów w programie. Jeżeli przyjęte zostaną następujące oznaczenia:

N — liczba wprowadzonych błędów,

M — liczba wszystkich wykrytych błędów,

X — liczba wprowadzonych błędów, które zostały wykryte,

to szacunkowa liczba błędów przed wykonaniem testów była następująca:

$$(M-X) \times N / X,$$

a liczba błędów po usunięciu wykrytych jest następująca:

$$(M-X) \times (N / X - 1).$$

To drugie wyrażenie zakłada oczywiście, że sztucznie wprowadzone błędy zostały usunięte z programu.

Powyższe oszacowania opierają się na założeniu, że wprowadzane błędy są podobne do błędów już znajdujących się w programie a poszczególne klasy wprowadzanych błędów występują w podobnych proporcjach. Jeżeli warunek ten nie jest spełniony, to szacunki liczby błędów mogą się okazać mocno chybione.

Technika posiewania błędów pozwala nie tylko oszacować liczbę błędów, lecz również ocenić efektywność stosowanych technik testowania. Stosunek X / N opisuje efektywność wykonanych testów. Jego zbyt mała wartość wskazuje na konieczność poprawy sposobów testowania

9.4. Testy systemu

Dwie główne techniki testów systemu to:

- *testowanie wstępujące*,
- *testowanie zstępujące*.

W przypadku testowania wstępującego najpierw testowane są pojedyncze moduły, następnie moduły coraz wyższego poziomu, aż do osiągnięcia poziomu całego systemu. Zastosowanie tego podejścia nie zawsze jest jednak możliwe gdyż poszczególne moduły mogą nie być niezależne od innych. Może więc być niemożliwe przetestowanie pojedynczych modułów bez odwoływania się do innych modułów. Możliwe jest oczywiście zastąpienie modułów, do których następują odwołania przez ich szkieletowe implementacje. W ten sposób łatwo jest jednak ominąć pewne błędy, które objawiają się jedynie wtedy, gdy następuje współpraca z w pełni działającym modułem.

Testowanie zstępujące rozpoczyna się od testowania modułów wyższego poziomu. Ich testowanie wymaga konstrukcji szkieletów modułów niższego poziomu. Po przetestowaniu modułu wyższego poziomu dołączane są moduły poziomu niższego i rozpoczynane są test tej grupy modułów. Proces ten kontynuuje się aż do zintegrowania i przetestowania całego systemu.

Omówione w poprzednich punktach testy koncentrowały się na wykrywaniu błędów i ocenie niezawodności. W fazie testów systemu wykonuje się także inne rodzaje testów o odmiennych celach.

- *Testy pod obciążeniem* (ang. *stress testing*). Celem tych testów jest zbadanie wydajności oraz niezawodności systemu podczas pracy pod pełnym lub nawet nadmiernym obciążeniem. Dotyczy to szczególnie systemów wielodostępnych i sieciowych. Systemy takie muszą spełniać pewne wymagania wobec wydajności. Muszą na przykład zapewniać obsługę określonej liczby użytkowników lub realizację podanej liczby transakcji w ciągu godziny. Omawiane testy polegają na wymuszeniu obciążenia równego lub nawet większego od maksymalnego. Testowanie pod przeciążeniem jest typowym podejściem stosowanym w innych dziedzinach techniki.
- *Testy odporności* (ang. *robustness testing*). Celem tych testów jest sprawdzenie działania w przypadku zajścia niepożądanych zdarzeń, na przykład:
  - zaniku zasilania,
  - awarii sprzętowej,
  - wprowadzenia niepoprawnych danych,
  - wydania sekwencji niepoprawnych poleceń.

9.5. Bezpieczeństwo oprogramowania

Pewne systemy komputerowe są krytyczne z punktu widzenia bezpieczeństwa ludzi. Oznacza to, że w przypadku takich systemów błędne wykonania mogą prowadzić do zagrożenia życia i zdrowia ludzkiego. Niebezpieczeństwo dla życia i zdrowia może być bezpośrednie, jeżeli system automatycznie steruje sprzętem ważnym dla życia lub zdrowia. Przykłady systemów, w których zachodzi zagrożenie bezpośrednie to oprogramowanie sterujące aparaturą medyczną i oprogramowanie wspomagające pilotowanie samolotów. Niebezpieczeństwo może być także pośrednie, jeżeli oprogramowanie doradza jedynie pewne działania człowiekowi. Przykładami takich systemów są systemy eksperckie stosowane w medycynie i bazy danych o lekach.

W przypadku systemów krytycznych z punktu widzenia bezpieczeństwa tylko niektóre błędne wykonania są rzeczywiście niebezpieczne. Bezpieczny system nie musi więc być w pełni niezawodny. Omawiane poniżej techniki zapewniania bezpieczeństwa oprogramowania opierają się na założeniu, że liczba błędów które mogą prowadzić do niebezpiecznych błędnych wykonań jest znacznie mniejsza od liczby wszystkich błędów w programie.

Techniki omawiane w tym rozdziale znajdują zastosowanie nie tylko w przypadku systemów krytycznych z punktu widzenia bezpieczeństwa. W wielu systemach można wyróżnić błędne wykonania, które mogą spowodować szczególnie niepożądane rezultaty, np. duże straty materialne lub złamanie przepisów prawnych. Proponowane podejście pozwala na unikanie szczególnie niepożądanych błędnych wykonań.

Należy podkreślić, że bezpieczeństwo oprogramowania, choć jest ściśle związane z niezawodnością, nie może być jednak z nią utożsamiane. Wynika to z następujących czynników:

- System zawodny może być bezpieczny, jeżeli skutki błędnych wykonań nigdy nie są groźne.
- Wymagania wobec systemu mogą być niepełne i nie opisywać zachowania systemu we wszystkich sytuacjach. Dotyczy to zwłaszcza sytuacji wyjątkowych, np. wprowadzenie niewłaściwych danych lub wydania niepoprawnego ciągu poleceń. Ważne jest aby system zachowywał się bezpiecznie także wtedy, gdy właściwy sposób reakcji nie został opisany.
- Niebezpieczeństwo może wynikać z awarii sprzętowych. Bezpieczeństwo oprogramowania wiąże się więc z jego odpornością na tego typu sytuacje.

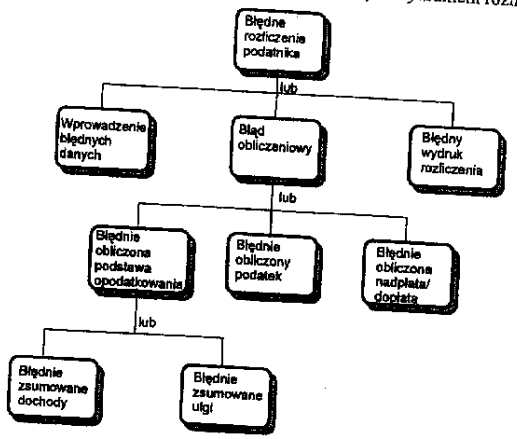
Analiza bezpieczeństwa rozpoczyna się od określenia potencjalnych niebezpieczeństw związanych z użytkowaniem systemu. Pod pojęciem niebezpieczeństwa rozumie się sytuację będącą wynikiem działania systemu, w której zachodzi ryzyko utraty życia, utraty zdrowia, dużych strat materialnych lub złamania przepisów prawnych. Jako przykł posłuży omawiany w książce program podatkowy. Niebezpieczeństwa jakie się wią z użytkowaniem tego systemu to:

- błędne rozliczenie podatnika z urzędem podatkowym,
- nie złożenie zeznania podatkowego,
- złożenie wielu zeznań dla jednego podatnika, być może w różnych urzędach.

W dalszym ciągu analizowane będzie pierwsze z tych niebezpieczeństw.

Podstawową techniką zwiększania bezpieczeństwa jest analiza drzew błędów (ang. *fault tree*). Korzeniem takiego drzewa jest jedna z rozważanych niebezpiecznych sytuacji. Wierzchołkami są pośrednie sytuacje, które mogą prowadzić do zaistnienia sytuacji odpowiadającej wierzchołkowi wyższego poziomu. Przedstawione na rysunku 9.3 drzewo błędów oznacza np., że błędne rozliczenie podatnika może być spowodowane wprowadzeniem błędnych danych, błędem obliczeniowym lub błędnym wydrukiem rozliczenia.

Rysunek 9.3. *Fragment drzewa błędów systemu podatkowego*



Drzewo błędów wskazuje możliwe przyczyny zajścia niebezpieczeństwa. Unikanie niebezpieczeństwa polega więc na unikaniu jego przyczyn. Techniki, które pozwalają na zmniejszenie ryzyka zajścia niebezpieczeństwa to:

- Położenie dużego nacisku na unikanie błędów podczas implementacji fragmentów systemu, w których błędy mogą prowadzić do niebezpieczeństwa. W omawianym przykładzie systemu podatkowego są to np. fragmenty odpowiedzialne za sumowanie dochodów, sumowanie ulg, obliczanie należnego podatku.
- Zlecenie realizacji powyższych fragmentów systemu najbardziej doświadczonym programistom.
- Zastosowanie techniki programowania N-wersyjnego w przypadku wymienionych fragmentów systemu (patrz rozdział 7.1.2 omawiający tolerancję błędów).

- Szczególnie dokładne przetestowanie tych fragmentów systemu.
- Wczesne wykrywanie sytuacji, które mogą być przyczyną niebezpieczeństwa i podjęcie odpowiednich, bezpiecznych akcji. W omawianym systemie taką akcją może być poinformowanie użytkownika i uniemożliwienie wydruku rozliczenia.

9.6. Podsumowanie

9.6.1. Kluczowe czynniki sukcesu

Główne czynniki wpływające na sukces fazy testowania to:

- *Określenie fragmentów systemu o szczególnych wymaganiach wobec niezawodności.*
- *Właściwa motywacja osób zaangażowanych w testowanie.* Praca nad testowaniem często nie jest ceniona na równi z pracą nad konstrukcją oprogramowania. Z tego powodu osoby, które zajmują się testowaniem często są uznawane za pracowników mniej ważnych. Osoby te mogą nawet spotykać się z nieprzychylnością innych pracowników jako ci, którzy wykazują błędy w ich pracy. Jeżeli testowaniem zajmują się osoby, które jednocześnie pracują jako programiści i projektanci, to często nie traktują oni testowania równie poważnie jak projektowania czy programowania. Zadaniem kierownictwa przedsięwzięcia jest więc właściwe motywowanie osób zajmujących się testami. Ważne jest wytworzenie wśród pracowników firmy przekonania o istotnym znaczeniu niezawodności oprogramowania, a co za tym idzie o ważnej roli testów i innych wysiłków zmierzających do poprawy niezawodności. Ważne jest także docenianie osób wykonujących testy, szczególnie tych osiągających najlepsze wyniki. Zaleca się nagradzanie osób, które wykryły najwięcej błędów lub wykryły najważniejszy błąd. Nagroda nie musi być zresztą nagrodą o dużej wartości materialnej. Często wystarczająca jest nagroda symboliczna lub przyznawanie tytułu „najlepszego łowcy błędów".

Wydaje się, że pewne osoby mają szczególny talent wykrywania błędów. Prawdopodobnie są to osoby o tendencjach do działania w sposób niestandardowy. Ponieważ wiele błędów wiąże się z nieuwzględnieniem sytuacji wyjątkowych, są one trudne do wykrycia przez osoby używające system w typowy sposób.

9.6.2. Podstawowe rezultaty fazy testowania

Podstawowe rezultaty fazy testowania to:

- poprawiony kod, projekt, model i specyfikacja wymagań,
- raport przebiegu testów, zawierający informacje o wykonanych testach i ich rezultatach,

- oszacowanie niezawodności oprogramowania i kosztów konserwacji.

9.6.3. Narzędzia CASE w fazie testowania

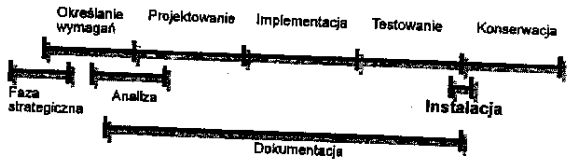
Testowanie odbywa się na podstawie specyfikacji wymagań zawartej w słowniku danych. Wykorzystuje się więc możliwości wyszukiwania informacji w słowniku danych i przygotowywania raportów.

W przypadku wykrycia błędów, których usunięcie wymaga poprawy projektu, modelu lub specyfikacji wymagań, wykorzystywane są oczywiście wszystkie możliwości narzędzi CASE wykorzystywane w poprzednich fazach.

Rozdział 10. Instalacja

10.1. Zadania wykonywane w fazie instalacji

Faza instalacji dotyczy oprogramowania wytwarzanego na zamówienie. W jej trakcie następuje przekazanie systemu użytkownikowi, który w momencie jej zakończenia stanie się właścicielem systemu.



Rysunek 10.1. Faza instalacji w cyklu życia oprogramowania

Na fazę instalacji składają się:

- szkolenia użytkowników końcowych i administratorów systemu,
- instalacja sprzętu i przeniesienie oprogramowania,
- wypełnienie baz danych,
- nadzorowane korzystanie z systemu, często równoległe z tradycyjnym sposobem pracy,
- usuwanie błędów w oprogramowaniu i dokumentacji użytkowej,
- przekazanie systemu klientowi.

Zaleca się, aby szkolenia użytkowników były wykonywane przez osoby, które były zaangażowane w prowadzenie przedsięwzięcia. Osobom tym będzie znacznie łatwiej nawiązać kontakt z uczestnikami szkoleń. Nie będą też posługiwać się językiem technicznym, niezrozumiałym dla uczestników szkoleń.

Wypełnianie bazy danych może być żmudnym procesem wymagającym długotrwałego wprowadzania danych przechowywanych w formie papierowej. Obecnie coraz częściej zdarza się, że system jest instalowany u klienta, który już wcześniej korzystał z innego oprogramowania. W tym wypadku znaczna część danych może być dostępna w formie elektronicznej. Ponieważ format przechowywania danych w nowym systemie będzie prawdopodobnie inny niż w poprzednim, niezbędna może być konstrukcja dodatkowego oprogramowania służącego do konwersji danych. Standardy dostępu do baz danych (na przykład ODBC) zdecydowanie ułatwiają konwersję danych. Konwersja danych będzie również znacznie łatwiejsza, jeżeli klient uzyska od dostawcy poprzedniego systemu specyfikację struktury bazy danych.

Planowanie i harmonogramowanie prac jest szczególnie istotne w fazie instalacji. W fazie tej często pojawia się szereg problemów technicznych oraz konieczność szybkiego usunięcia zauważonych błędów i wprowadzenia modyfikacji. Analitycy, projektanci i programiści systemu muszą więc być na bieżąco dostępni. Z drugiej strony z reguły nie mogą oni zarezerwować całego swojego czasu na prace związane z instalacją. Podobne problemy powstają także po stronie klienta. W trakcie instalacji użytkownicy nie mogą zaniechać realizacji swoich podstawowych zadań.

Producenci oprogramowania powinni być przygotowani na pewien opór klienta związany z koniecznością zmiany sposobu pracy. Przedstawiciele klienta, z którymi stykano się w poprzednich fazach to często osoby o znacznie lepszym przygotowaniu informatycznym i znacznie lepiej nastawione do instalowanego systemu. Ważne jest uzyskanie wstępnej akceptacji osób, które po raz pierwszy zetknęły się z systemem.

10.2. Podsumowanie

10.2.1. Kluczowe czynniki sukcesu

Główne czynniki wpływające na sukces fazy instalacji to:

- *Właściwe planowanie i harmonogramowanie zadań*, które zapewni szybki przebieg instalacji przy jednoczesnym niezakłócaniu innych prac klienta i producenta.
- *Uzyskanie wstępnej pozytywnej reakcji użytkowników.*

10.2.2. Podstawowe rezultaty fazy instalacji

Podstawowe rezultaty fazy instalacji to:

- poprawiony kod, projekt, model i specyfikacja wymagań,
- oprogramowanie zainstalowane u użytkownika.

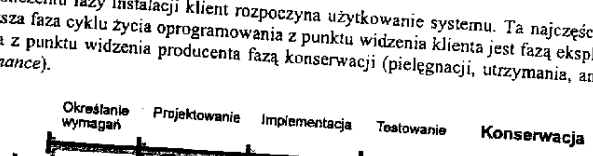
10.2.3. Narzędzia CASE w fazie instalacji

W przypadku wykrycia błędów lub konieczności dokonania zgłoszonych przez klienta modyfikacji wykorzystywane są te same możliwości narzędzi CASE co w poprzednich fazach.

Rozdział 11. Konserwacja oprogramowania

Po zakończeniu fazy instalacji klient rozpoczyna użytkowanie systemu. Ta najczęściej najdłuższa faza cyklu życia oprogramowania z punktu widzenia klienta jest fazą eksploatacji, a z punktu widzenia producenta fazą konserwacji (pielęgnacji, utrzymania, ang. *maintanance*).

sunek 11.1. a konserwacji yklu życia ogramowania



11.1. Modyfikowanie oprogramowania

Konserwacja oprogramowania polega na wprowadzaniu modyfikacji. Istnieją trzy główne klasy wprowadzanych w oprogramowaniu modyfikacji:

- *Modyfikacje poprawiające* polegają na usuwaniu z oprogramowania błędów.
- *Modyfikacje ulepszające* polegają na poprawie jakości oprogramowania.
- *Modyfikacje dostosowujące* polegają na dostosowaniu oprogramowania do zmian zachodzących w środowisku jego pracy.

Poprawa może dotyczyć błędów popełnionych w fazach określania wymagań, analizy, projektowania i implementacji. Oczywiście im wcześniejsza jest faza, w której popełniono błąd, tym większy jest spodziewany koszt poprawny tego błędu. Przykłady modyfikacji ulepszających to:

- poprawa wydajności pewnych funkcji,
- poprawa ergonomii interfejsu użytkownika,
- poprawa przejrzystości raportów.

Modyfikacje dostosowujące mogą przykładowo wynikać z:

- zmiany wymagań użytkowników,
- zmian przepisów prawnych dotyczących dziedziny problemu,
- zmian organizacyjnych po stronie klienta,
- zmian sprzętu i oprogramowania systemowego.

Część modyfikacji prowadzi jedynie do niewielkich zmian kodu, które nie wymagają zmiany dokumentacji technicznej. Przykładem są typowe błędy programistyczne, np. błąd w instrukcji warunkowej. Inne modyfikacje prowadzą do przebudowy oprogramowania, która musi zostać uwzględniona w dokumentacji technicznej. Osoby wykonujące modyfikacje często mają tendencję do szybkiego dokonania modyfikacji polegającego jedynie na zmianie kodu. Skraca to nieznacznie czas niezbędny na wprowadzenie modyfikacji, prowadzi jednak do sytuacji, w której dokumentacja techniczna nie odpowiada rzeczywistemu systemowi. Dlatego zaleca się, aby wprowadzanie modyfikacji polegało na powrocie do najwcześniejszej z faz, na której rezultaty wpływa dana zmiana w oprogramowaniu. Jeżeli np. modyfikacja nie zmienia wymagań wobec systemu, lecz zmienia związki pomiędzy klasami składowej dziedziny problemu, to należy zmodyfikować model, projekt i kod. Narzędzia CASE ułatwiają dokonywanie takich modyfikacji redukując nakład czasu niezbędny na zmiany modelu i projektu.

Przyczynami wprowadzania modyfikacji są z reguły żądania użytkowników. Kierownictwo firmy może także zdecydować się na wprowadzanie zmian na podstawie porównania możliwości oferowanego oprogramowania z ofertą konkurentów.

Żądana zmiana przed jej ewentualnym wprowadzenie powinna zostać oceniona pod kątem sensowności jej wprowadzania. Pierwszym krokiem powinna być zatem analiza wpływu wprowadzenia danej zmiany. Analiza ta powinna uwzględniać:

- znaczenie wprowadzenia zmiany dla użytkowników,
- koszt wprowadzenia zmiany,
- wpływ zmiany za poszczególne składowe systemu,
- wpływ zmiany na poszczególne składowe dokumentacji technicznej.

Dopiero po dokonaniu oceny zmiany podejmowana jest decyzja o jej ewentualnej realizacji. Decyzje takie powinna podejmować jedna uprawniona osoba. W przypadku bardzo dużych przedsięwzięć może zostać powołana specjalna komisja uprawniona do podejmowania decyzji o realizacji zmian.

Zaakceptowane zmiany nie są realizowane natychmiast. Zaleca się raczej grupowanie zmian, których wykonanie prowadzi do powstania nowej wersji systemu.

Podobnie jak ma to miejsce w przypadku testowania, praca nad konserwacją oprogramowania jest często ceniona niżej niż jego wytwarzanie. Zadaniem kierownictwa jest więc właściwe motywowanie i docenianie osiągnięć osób pracujących nad wprowadzaniem zmian.

Koszty konserwacji mogą być bardzo duże. Często przekraczają znacznie koszt opracowania systemu. Niedocenianie nakładów pracy na fazę konserwacji jest jedną z głównych przyczyn opóźnień przedsięwzięć programistycznych (van Genuchtena, 1991).

Pewne czynniki wpływające na koszty konserwacji można uznać za obiektywne, to jest takie, na które kierownictwo przedsięwzięcia ma znikomy wpływ. Do czynników takich można zaliczyć:

- *Stabilność środowiska, w którym pracuje system.* Zmiany zachodzące w przepisach prawnych, zmiany struktury organizacyjnej i sposobów działania po stronie klienta prowadzą do zmian wymagań wobec systemu.
- *Stabilność platformy sprzętowej i oprogramowania systemowego.*
- *Czas użytkowania systemu.* Całkowite koszty konserwacji rosną oczywiście wtedy, gdy system jest użytkowany przez długi okres.

Na redukcję kosztów konserwacji wpływają następujące czynniki:

- *Znajomość dziedziny problemu.* Jeżeli analitycy pracujący nad systemem dobrze znają daną dziedzinę problemu, mają mniej trudności z właściwym zebraniem wymagań oraz budową oddającego rzeczywistość modelu.
- *Wysoka jakość modelu i projektu,* w szczególności (porównaj sekcję 6.7 omawiającą jakość projektu):
  - spójność,
  - stopień powiązań składowych,
  - przejrzystość.
- *Wysoka jakość dokumentacji technicznej.* Dokumentacja techniczna powinna:
  - w pełni odpowiadać systemowi,
  - być wystarczająco szczegółowa,
  - być zgodna z przyjętymi w firmie standardami.

Dalsze zagadnienia związane z dokumentacją techniczną omówiono w sekcji 13.5.2.

- *Stabilność personelu.* Wysoka jakość projektu oraz dokumentacji technicznej nie zmienia faktu, że pewne aspekty systemu zawsze są najlepiej znane osobom bezpośrednio uczestniczącym w jego realizacji. Nie oznacza to automatycznie, że modyfikacje powinny być zawsze wykonywane przez te same osoby, które opracowywały system. Jeżeli jednak osoby te nadal pracują u producenta oprogramowania, to istnieje możliwość konsultacji w przypadku pojawienia się istotnych wątpliwości i trudności.

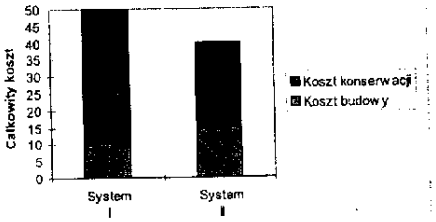
- *Środowisko implementacji.* Zaawansowane środowiska implementacji, obejmujące języki wysokiego poziomu i możliwości dialogowego projektowania i wytwarzania fragmentów oprogramowania (na przykład narzędzia RAD) wpływają na skrócenie czasu niezbędnego na wprowadzenie modyfikacji.

- *Niezawodność oprogramowania.* Wysoka niezawodność oprogramowania w momencie przekazania do użytkowania zmniejsza liczbę modyfikacji, w szczególności o charakterze poprawiającym.

- *Inżynieria odwrotna.* Pod tym pojęciem rozumie się odtwarzanie dokumentacji technicznej na podstawie istniejącego oprogramowania. Inżynieria odwrotna omawiana jest w sekcji 11.2.

- *Zarządzanie wersjami.* Zagadnienia te omawiane są w sekcji 13.6.

Jak wynika z powyższej listy, wiele działań zmierzających do redukcji kosztów konserwacji musi być podejmowanych już w trakcie budowy systemu. Działania takie mogą prowadzić do wzrostu kosztów ponoszonych w trakcie realizacji oprogramowania. Ponieważ jednak całkowity koszt konserwacji może znacznie przekraczać koszt budowy, większe koszty poniesione w trakcie realizacji systemu powinny spowodować obniżenie całkowitego kosztu oprogramowania (porównaj rysunek 11.2).



Rysunek 11.2. *Przykładowy wpływ kosztów poniesionych w trakcie budowy systemu na jego całkowity koszt*

Jako przykład wprowadzania modyfikacji zostanie omówiony problem wprowadzenia następujących zmian do systemu informacji geograficznej, omawianego w poprzednich rozdziałach książki (sekcja 5.4.3.2 omawia konstrukcję modelu dla tego systemu):

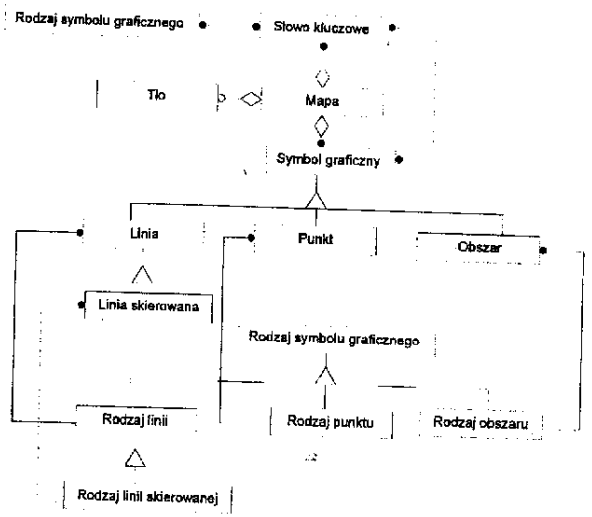
Żądane zmiany w systemie informacji geograficznej

1. W wielu przypadkach na mapie umieszcza się symbole, które powinny wyglądać w podobny sposób. Na przykład wszystkie symbole punktowe opisujące stacje benzynowe oznaczane są tym samym symbolem, wszystkie linie autobusowe oznaczane są linią tego samego rodzaju. Przydatna byłaby więc możliwość definiowania rodzajów symboli, oznaczanych na mapie w ten sam sposób. Wprowadzając nowy symbol należałoby wtedy wskazać do jakiego rodzaju symboli ma on należeć.

2. Niezbędne jest wprowadzenie nowego rodzaju symbolu — linii skierowanej, która pozwalałby na przykład przedstawiać na mapie trasy dojścia/dojazdu z pewnego miejsca do punktu docelowego.

Uwzględnienie tych zmian prowadzi do modelu klas przedstawionego na rysunku 11.3.

Rysunek 11.3. *Zmodyfikowany model klas systemu informacji geograficznej*



Po wprowadzeniu powyższych zmian pola opisujące sposób wyświetlania poszczególnych symboli umieszczane są w klasach opisujących rodzaje symboli.

Zmodyfikowany system informacji geograficznej — pola klas

Mapa

Nazwa — ciąg znaków

Wyświetlany obszar — prostokąt (dana złożona)

Tło

Nazwa pliku — identyfikator pliku

Słowo kluczowe

Nazwa — ciąg znaków

Opis — ciąg znaków

Stan — wybrane lub niewybrane

Symbol graficzny

Nazwa — ciąg znaków

Opis — polecenie systemu operacyjnego wywołujące zewnętrzny program wyświetlający opis symbolu

Punkt

Współrzędna pozioma — liczba całkowita

Współrzędna pionowa — liczba całkowita

Linia

Lista punktów — lista współrzędnych punktów tworzących linię (dana złożona)

Obszar

Lista punktów — lista współrzędnych punktów tworzących linię ograniczającą obszar (dana złożona)

Rodzaj punktu

Znak — nazwa pliku zawierającego symbol punktu w formacie WMF (Windows Metafile)

Rodzaj linii

Wzór — identyfikator wzoru linii

Kolor — identyfikator koloru linii

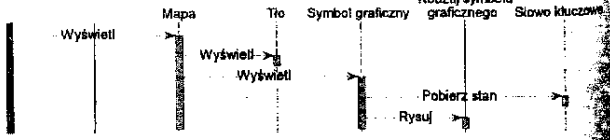
Rodzaj obszaru

Wypełnienie — identyfikator wypełnienia obszaru

Kolor — identyfikator koloru obszaru

Model klas systemu uległ dość znacznej rozbudowie. Okazuje się jednak, że wpływa to tylko w niewielkim stopniu na scenariusze przepływu komunikatów w systemie. Rysunek 11.4 przedstawia scenariusz przepływu komunikatów podczas wyświetlania mapy (porównaj rysunek 5.58 przedstawiający ten scenariusz dla oryginalnego systemu).

Rysunek 11.4. Zmodyfikowany scenariusz wyświetlania mapy



Najistotniejsze jest, że scenariusz ten nie wprowadza konieczności dokonywania zmian w klasie mapa, która nadal wysyła jedynie komunikat Wyświetl do klasy Symbol g

ficzny. Jest więc to przykład korzyści na etapie konserwacji będących rezultatem niezależności składowych oraz wykorzystania polimorfizmu metod.

11.2. Inżynieria odwrotna

W wielu przypadkach zachodzi konieczność wprowadzenia zmian w oprogramowaniu, które nie jest opisane dokumentacją techniczną. Inżynieria odwrotna (ang. *reverse engineering*) jest procesem polegającym na odtworzeniu dokumentacji technicznej na podstawie istniejącego kodu (czasami także na podstawie struktury bazy danych). Proces ten nazywany jest w ten sposób, gdyż jest on swego rodzaju odwróceniem kolejności czynności wykonywanych w trakcie budowy oprogramowania. W fazie implementacji wykonywana jest transformacja projektu do kodu. Należy przy tym pamiętać, że opis projektu staje się po zakończeniu implementacji dokumentacją techniczną. Inżynieria odwrotna jest więc próbą odtworzenia projektu, na bazie którego mogłoby powstać istniejące oprogramowanie.

Podczas próby odtworzenia dokumentacji technicznej na podstawie kodu może się okazać, że wiele konstrukcji zastosowanych w programie nie da się zapisać w przyjętej notacji. Dotyczy to w szczególności programów pisanych w językach hybrydowych, np. C++ z zastosowaniem zarówno konstrukcji obiektowych, jak i strukturalnych.

Inżynieria odwrotna jest procesem, który daje się automatyzować. Moduły inżynierii odwrotnej są więc częstymi składowymi narzędzi CASE. Pewne składowe projektu mogą być jednak implementowane w różny sposób (porównaj rozdział 6.1, omawiający uszczegóławianie wyników analizy). Przykładowo związki klas implementuje się wprowadzając do powiązanych klas dodatkowe pola, które mogą być obiektami powiązanej klasy, wskaźnikami do obiektów powiązanej klasy lub identyfikatorami obiektów powiązanej klasy. Dodatkowo w przypadku związku z wieloma obiektami pojawiają się różne możliwości stosowania złożonych struktur danych. Narzędzia inżynierii odwrotnej mogą więc wymagać właściwego skonfigurowania. Może też okazać się, że automatycznie odtworzona dokumentacja będzie wymagała dalszej ręcznej korekty.

Typowym przykładem inżynierii odwrotnej jest odtwarzanie modelu związków encji na podstawie struktury bazy danych. Poniższy przykład zawiera zestaw instrukcji w języku SQL, na podstawie którego możliwe jest odtworzenie modelu przedstawionego na rysunku 11.5. W ramach technik strukturalnych wykorzystuje się także informacje o nagłówkach procedur i funkcji oraz ich wywołaniach dla odtworzenia informacji o modułach i ich związkach.

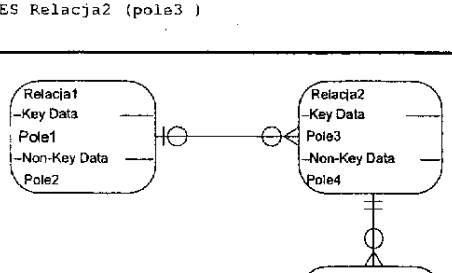
```
CREATE TABLE Relacja1 (
     Pole1 TEXT NOT NULL,
     Pole2 TEXT NOT NULL,
     PRIMARY KEY ( Pole1 )
```

```
CREATE TABLE Relacja2 (
     pole3 TEXT NOT NULL,
     pole4 DATE NOT NULL,
     Pole1 TEXT NOT NULL,
     PRIMARY KEY ( pole3 )
)

ALTER TABLE Relacja2 (
     ADD FOREIGN KEY (Pole1 )
     REFERENCES Relacja1 (Pole1 )
        ON DELETE CASCADE
)
CREATE TABLE Relacja3 (
     pole5 TEXT NOT NULL,
     pole6 DECIMAL NOT NULL,
     pole3 TEXT NOT NULL,
     PRIMARY KEY ( pole5 )
)

ALTER TABLE Relacja3 (
     ADD FOREIGN KEY (pole3 )
     REFERENCES Relacja2 (pole3 )
)
```

Rysunek 11.5.
Odtworzony model związków encji



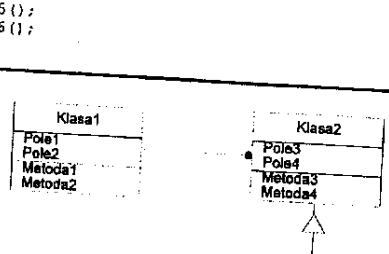
W przypadku technik obiektowych stosunkowo trudniejsze jest odtworzenie informacji o związkach klas, gdyż należy brać pod uwagę rozmaite sposoby ich implementacji. Poniższy przykład zawiera fragment kodu w języku C++, na podstawie którego można odtworzyć model związków klas z rysunku 11.6.

```
class Klasa1
{
private:
    long Pole1;
    double Pole2;
public:
    void Metoda1();
    void Metoda2();
protected:
    vector<Klasa2*> vKlasa2;
};

class Klasa2
{
private:
    unsigned short Pole3;
    float Pole4;
public:
    void Metoda3();
    void Metoda4();
protected:
    Klasa1* pKlasa1;
};

class Klasa3
                    : public Klasa2
{
private:
    long Pole5;
    double Pole6;
public:
    void Metoda5();
    void Metoda6();
};
```

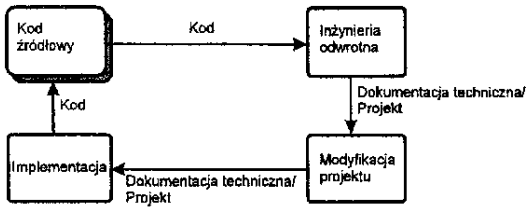

Rysunek 11.6.
Odtworzony model klas



Odtworzona dokumentacja techniczna może zostać zmodyfikowana. Jeżeli następnie na jej podstawie modyfikowany jest kod, to proces taki nazywany jest reinżynierią oprogramowania (ang. *software re-engineering*, porównaj rysunek 11.7).

Faza konserwacji jest ostatnim etapem przedsięwzięcia programistycznego. Po pewnym czasie użytkowania systemu narasta potrzeba dokonania tak istotnych modyfikacji, że możliwe są dwa wyjścia:

- zaprzestanie korzystania z systemu,
- rozpoczęcie prac nad nową wersją, czyli rozpoczęcie nowego przedsięwzięcia.



Rysunek 11.7. Reinżynieria oprogramowania

11.3. Podsumowanie

11.3.1. Kluczowe czynniki sukcesu

Najistotniejsze elementy wpływające na sukces fazy konserwacji to:

- *wysoka jakość definicji wymagań, modelu i projektu,*
- *dobra znajomość środowiska implementacji,*
- *właściwa motywacja osób wykonujących konserwację oprogramowania,*
- *właściwe oszacowanie kosztów konserwacji.*

11.3.2. Podstawowe rezultaty fazy konserwacji

Podstawowe rezultaty fazy konserwacji to:

- poprawione kod, projekt, model i specyfikacja wymagań.

11.3.3. Narzędzia CASE w fazie konserwacji

Wprowadzenie żądanych modyfikacji może wymagać cofnięcia się nawet do fazy określania wymagań. Dlatego też w fazie konserwacji wykorzystywane są te same składowe narzędzi CASE, które były stosowane we wszystkich poprzednich fazach.

Jak już wspomniano narzędzia CASE wspomagają proces inżynierii odwrotnej. Pozwalają one na podstawie istniejącego kodu odtworzyć dokumentację techniczną. Wyniki pracy narzędzi inżynierii odwrotnej są umieszczane w słowniku danych. Czasami możliwe jest także (pół)automatyczne utworzenie diagramów graficznych na podstawie odtworzonej zawartości słownika danych.

Narzędzia CASE pozwalają oczywiście na dalsze modyfikacje odtworzonego projektu i wygenerowanie kodu, wspomagają więc także proces reinżynierii oprogramowania.

Rozdział 12. Narzędzia CASE

12.1. Rodzaje narzędzi CASE

Termin CASE (ang. *computer assisted/aided software/system engineering*) oznacza komputerowo wspomaganą inżynierię oprogramowania/systemów. Nazwa ta sugeruje, że pod hasłem tym kryją się wszelkie narzędzia komputerowe wykorzystywane w trakcie prac na oprogramowaniem, a więc także kompilatory, debuggery, edytory tekstu, narzędzia harmonogramowania przedsięwzięć, arkusze kalkulacyjne. Tradycyjnie jednak pod pojęciem CASE rozumie się narzędzia, które:

- wspomagają ogólnie rozumiane wytwarzanie oprogramowania,
- koncentrują się na fazach analizy i projektowania oraz bezpośrednim wykorzystaniu wyników tych faz w implementacji.

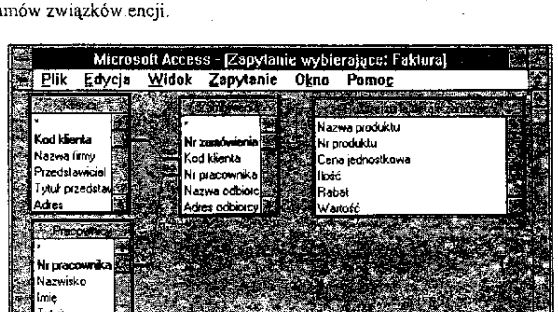
Jak już wspomniano w sekcji 1.3 wstępnie omawiającej narzędzia CASE, dzielą się one na dwie główne grupy:

- *Upper-CASE*, koncentrujące się na wspomaganiu wczesnych faz pracy nad oprogramowaniem, w szczególności fazy analizy. Narzędzia te nie są związane z konkretnym środowiskiem implementacji.
- *Lower-CASE*, koncentrujące się na fazach projektowania i implementacji. Narzędzia te są z reguły ściśle związane z konkretnym środowiskiem implementacji.

Narzędzia Upper-CASE powstały w odpowiedzi na potrzeby analityków i projektantów wysokiego poziomu, którzy zmagali się z analizą i projektowaniem złożonych systemów, lecz z reguły nie byli zaangażowani bezpośrednio w implementację systemu. Narzędzia Lower-CASE powstały w odpowiedzi na potrzeby projektantów niskiego poziomu oraz programistów. Chyba każdy kto pracował nad implementację dużego oprogramowania zgodzi się, że zapis kodu dużego programu w postaci tekstu nie ułatwia panowania nad całością systemu. Posługiwanie się notacjami graficznymi, w których poszczególne sym-

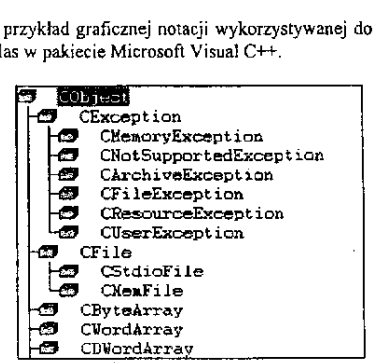
bole odpowiadają dającym się logicznie wyróżnić konstrukcjom programistycznym, oraz dialogowa edycja atrybutów tych konstrukcji wydaje się po prostu bardziej naturalna.

Warto zauważyć, że proces powstawania narzędzi Lower-CASE można obecnie obserwować w wielu pakietach programistycznych, w których pojawiają się elementy graficznej wizualizacji. Rysunek 12.1 zawiera przykład graficznej notacji służącej do projektowania zapytań w pakiecie Microsoft Access. Widoczne jest podobieństwo do notacji diagramów związków encji.



Rysunek 12.1. Graficzny sposób projektowania zapytań w pakiecie Microsoft Access

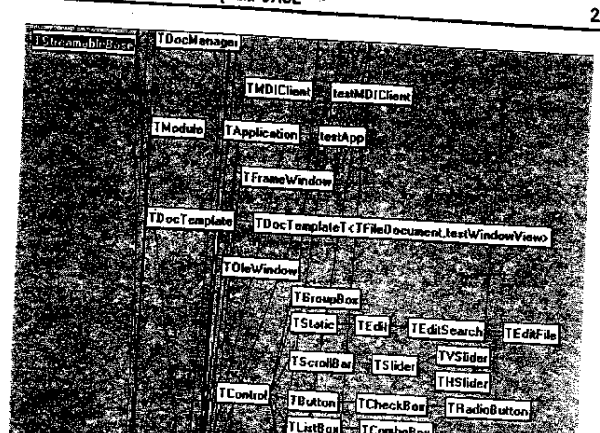
Rysunek 12.2 zawiera przykład graficznej notacji wykorzystywanej do prezentacji hierarchii dziedziczenia klas w pakiecie Microsoft Visual C++.



Rysunek 12.2. Graficzny sposób prezentacji hierarchii dziedziczenia klas w pakiecie Microsoft Visual C++

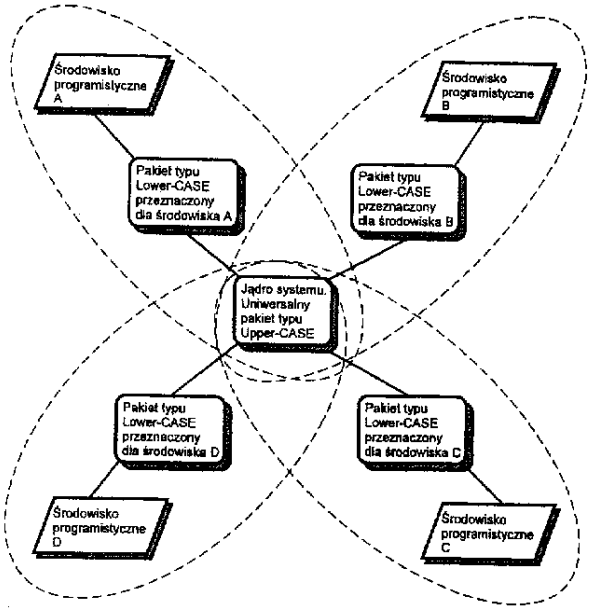
Podobny sposób prezentacji stosowany jest w pakiecie Borland C++ (porównaj rysunek 12.3). Pakiet Sun Works wspomagający programowanie w C++ pozwala na odtworzenie na podstawie analizy działania programu grafu wywołań, który prezentuje przepływ komunikatów pomiędzy obiektami w sposób bardzo zbliżony do diagramów interakcji. Wymienione notacje graficzne służą najczęściej jedynie do wizualizacji kodu programu, nie pozwalając na jego dialogową edycję. Można je więc jedynie traktować jako wstępny krok w stronę narzędzi Lower-CASE.

Rysunek 12.3. Graficzny sposób prezentacji hierarchii dziedziczenia klas w pakiecie Borland C++



Obecnie większość producentów pakietów CASE określa swoje produkty jako narzędzia I-CASE (Integrated-CASE). Są to pakiety łączące w sobie możliwości narzędzi Upper i Lower-CASE. Ponieważ jednak narzędzia Lower-CASE są związane z konkretnym środowiskiem implementacji, także pakiety I-CASE nie mogą być już tak uniwersalne jak narzędzia Upper-CASE. Niektórzy producenci starają się uniknąć tego problemu oferując systemy, których jądrem są uniwersalne narzędzia typu Upper-CASE. Dodatkowo producent oferuje kilka dodatkowych modułów typu Lower-CASE przeznaczonych dla różnych środowisk programistycznych (porównaj rysunek 12.4). Tym samym są oni w stanie zaoferować nabywcom kilka pakietów I-CASE dostosowanych do rozmaitych środowisk implementacji. Systemy takie można nazwać wielośrodowiskowymi narzędziami I-CASE. Produkcja takich systemów pozwala stosunkowo niewielkim kosztem poszerzyć ofertę dostawcy narzędzi CASE. Systemy takie są szczególnie atrakcyjne dla firm programistycznych wytwarzających oprogramowanie z wykorzystaniem rozmaitych środowisk lub planujących zmianę środowiska implementacji.

Narzędzia CASE mogą wspomagać jedną metodykę analizy i projektowania. Oferują one spójny zestaw notacji, który pozwala na stosowanie tej metodyki. Przykłady takich narzędzi to: Oracle CASE, EasyCASE, CASE 4.0, objectiF, Select OMT Professional. Nie musi to oznaczać, że narzędzia te trzymają się ściśle jednej ze standardowych metodyk znanych z literatury. Producenci pakietów Oracle CASE i Select OMT Professional proponują własne podejścia łączące elementy znane ze standardowych propozycji. Inne narzędzia pozwalają na stosowanie całego szeregu różnych notacji. Przykładami są pakiety System Architect, który wspomaga szereg notacji strukturalnych i obiektowych oraz Paradigm Plus, który wspomaga szeroki zakres różnorodnych notacji obiektowych.



Rysunek 12.4. Wielośrodowiskowe narzędzie I-CASE

Narzędzia typu I-CASE uzupełnione o narzędzia wspomagające zarządzanie przedsięwzięciem programistycznym takie, jak:

- narzędzia harmonogramowania i monitorowania przedsięwzięć,
- narzędzia szacowania kosztów,
- narzędzia analizy decyzyjnej,

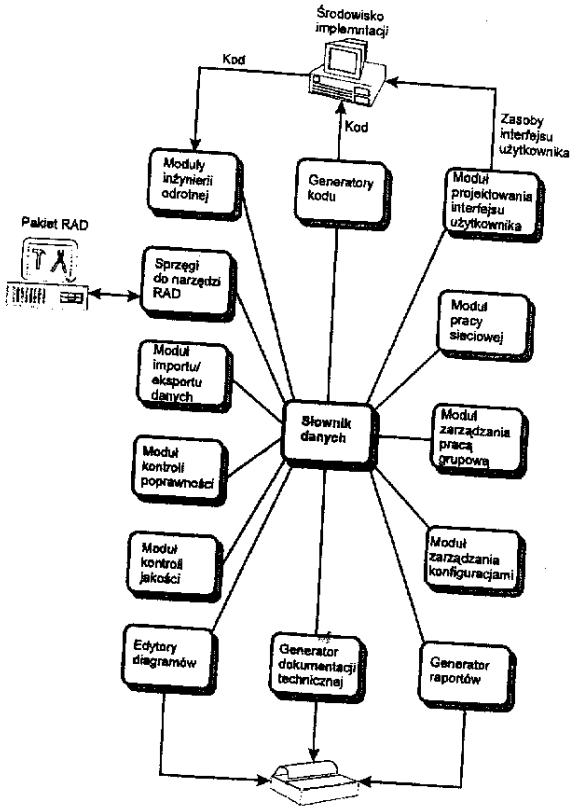
nazywane są *środowiskami inżynierii oprogramowania* (ang. *software engineering environments*).

12.2. Składowe narzędzi CASE

Rozdział ten zawiera charakterystykę poszczególnych składowych narzędzi CASE. Podkreślone są te cechy poszczególnych składowych, które mają istotny wpływ na ocenę poszczególnych systemów.

Wymienione poniżej składowe narzędzi CASE nie we wszystkich pakietach są wyraźnie wyodrębnionymi modułami. W wielu przypadkach funkcje kontroli poprawności są integralną częścią słownika danych i edytora diagramów. Często słownik danych i edytor diagramów stanowią jeden moduł, który stanowi jądro pakietu. W innych przypadkach wymienione składowe są jednak niezależnymi modułami.

Rysunek 12.5. Struktura narzędzi CASE



Edytor(y) notacji graficznych

Edytory notacji graficznych są to specjalizowane programy graficzne, pozwalające na realizację następujących funkcji:

- tworzenie i edycja diagramów wykorzystywanych w fazach określania wymagań, analizy i projektowania,
- tworzenie i edycja powiązań pomiędzy poszczególnymi symbolami i diagramami (np. powiązań pomiędzy symbolem procesu a diagramem, do którego dany proces jest dekomponowany) oraz nawigowanie po sieci powiązanych diagramów,
- wydruk diagramów.

Na ocenę edytorów notacji graficznych wpływają między innymi następujące czynniki:

- Ergonomia pracy. Diagramy graficzne są jednym z podstawowych narzędzi pracy w fazach analizy i projektowania. Powinny one pozwalać analitykom i projektantom skupić się na jego pracy, a nie na „zmaganiach" z edytorem.
- Możliwości kontrolowania ilości informacji prezentowanej w sposób graficzny.
- Jakość i możliwości formatowania wydruków.
- Wykrywanie na bieżąco niepoprawnych w danej notacji konstrukcji.
- Zapewnianie spójności informacji umieszczonych na różnych diagramach.

Słownik danych

Słownik danych (ang. *data dictionary*) lub repozytorium (ang. *repository*) oznacza nie tylko bazę danych zawierającą informacje o realizowanym projekcie, lecz także narzędzia służące do jego edycji i przeglądania. Podstawowe funkcje słownika danych to:

- wprowadzanie oraz edycja specyfikacji modelu i projektu, a także innych informacji związanych z przedsięwzięciem,
- wyszukiwanie pożądanej informacji.

W większości profesjonalnych narzędzi CASE dane zawarte w repozytorium przechowywane są w sposób, który czyni je dostępnym dla programów zewnętrznych. Z reguły dane te przechowywane są w bazie danych dostępnej dla innych programów. W przypadku pakietu System Architect jest to np. baza zgodna z formatem DBase III. Pakiet Paradigm Plus pozwala na wybór jednej z kilku baz danych w tym każdej zgodnej ze standardem ODBC. Podobne możliwości ma użytkownik pakietu Visible Analyst Workbench. Inna strategia jest zastosowana w pakiecie Select OMT Professional, w którym dostęp do słownika danych zapewnia mechanizm OLE Automation. Niektóre pakiety udostępniają specjalne sprzęgi programistyczne, które pozwalają na dostęp do informacji zawartych w słowniku danych. W przypadku pakietów StP (ang. *Software through Pictures*) jest to język zapytań SQL. W pakiecie Paradigm Plus jest to specjalny język skryptowy zbliżony do języka Visual Basic.

W wielu systemach użytkownik ma możliwość konfigurowania struktury słownika danych. W przypadku pakietu System Architect możliwości te są bardzo rozbudowane.

Użytkownik może konfigurować strukturę słownika danych edytując tekstowy plik konfiguracyjny. Istnieje tylko bardzo ograniczony stały zbiór atrybutów, opisujących poszczególne elementy słownika danych. Są to: nazwa, czas, data oraz identyfikator użytkownika, który dokonał ostatniej modyfikacji. Wszystkie pozostałe atrybuty opisu mogą być modyfikowane przez użytkownika. Oczywiście pewne z tych atrybutów mogą być wykorzystywane przez inne składowe pakietu, np. generatory kodu, więc ich usunięcie ze słownika danych może uniemożliwić pracę tych składowych. Użytkownik może dodawać także zupełnie nowe typy elementów umieszczanych w słowniku danych. Podobne możliwości ma na przykład pakiet Select OMT Professional. W tym wypadku nie można jednak dodawać do słownika nowych klas elementów, a liczba stałych atrybutów jest większa. W pakiecie tym można wprowadzać atrybuty, które nie będą widoczne dla użytkownika korzystającego ze standardowych narzędzi edycji słownika danych. Atrybuty takie mogą być wykorzystywane przez zewnętrzne moduły za pośrednictwem sprzęgu programistycznego.

Słownik danych należy uznać za podstawową składową narzędzi CASE. W wielu systemach możliwe jest pełne zdefiniowanie wymagań, modelu i projektu wyłącznie z poziomu słownika danych bez odwoływania się do diagramów graficznych. Jak wspomniano w sekcji 5.7.3 omawiającej zastosowanie narzędzi CASE w fazie analizy, w profesjonalnych narzędziach CASE diagramy graficzne są wygodnym narzędziem, służącym wizualizacji i edycji informacji zawartych w słowniku danych.

Słownik danych i edytor notacji graficznych stanowią często najprostszą konfigurację w jakiej można nabyć pakiety CASE.

Moduł kontroli poprawności

Pewne błędy mogą być wykrywane na bieżąco w trakcie edycji diagramów i słownika danych. Przykład takiego błędu to próba uczynienia klasy swoją specjalizacją lub generalizacją. Wykrycie innych błędów wymaga uruchomienia specjalnych funkcji odpowiedzialnych za weryfikację poprawności. Zagadnienia poprawności projektu omówione zostały w sekcji 6.6.

Moduł kontroli jakości

Pewne systemy zawierają funkcje pozwalające na automatyczną ocenę pewnych miar jakości projektu. Dotyczy to przede wszystkim złożoności oraz stopnia powiązań składowych. Zagadnienia jakości projektu omówione zostały w sekcji 6.7.

Generator raportów

Jest to moduł służący do przygotowywania raportów na podstawie zawartości słownika danych. Niektóre narzędzia obejmują raporty parametryczne, w przypadku których zakres i rodzaj informacji zależy od podanego parametru(ów). Przykładem może być raport, który otrzymuje jako parametr nazwę diagramu i obejmuje specyfikację symboli zawartych na tym diagramie. Narzędzia CASE dostarczane są z reguły ze sporym zbiorem gotowych raportów. Wiele systemów pozwala na definiowane własnych raportów.

Generator dokumentacji technicznej

Jest to moduł służący do przygotowywania dokumentacji technicznej złożonej z szeregu diagramów, raportów oraz dodatkowych opisów tekstowych. Moduł taki powinien pozwalać na swobodne formatowanie dokumentu oraz osiągnięcie wysokiej jakości wydruku. Moduł taki może być dostarczany ze zbiorem przykładowych dokumentów, na przykład zawierających informacje opisujące wyniki fazy analizy i projektowania. Użytkownik powinien mieć także możliwość definiowania własnych dokumentów o określonej przez niego zawartości. Generator dokumentacji technicznej powinien pozwalać na łatwe i efektywne uaktualnienie dokumentu po dokonaniu zmian w projekcie.

Generator(y) kodu

Są to narzędzia służące do generowania szkieletu kodu w rozmaitych językach programowania. Pozwalają one użytkownikowi uzupełnić specyfikację poszczególnych składowych o elementy specyficzne dla konkretnego języka programowania, np. określenie typów danych w konkretnym języku programowania (porównaj rozdział 6.1, omawiający uszczegółowienie wyników analizy). Wygenerowany kod może być uzupełniony o dodatkowe informacje zawarte w słowniku danych (np. słowny opis), umieszczane w wygenerowanym kodzie w formie komentarzy. Narzędzia takie pozwalają także użytkownikowi na określenie plików, w których ma być umieszczony kod wygenerowany dla poszczególnych składowych. Wygenerowane pliki powinny być automatycznie uzupełniane o odwołania do innych modułów. Konieczność wprowadzenia tych odwołań powinna być wyprowadzana na podstawie opisu projektu. Przydatna jest także możliwość czasowego zaniechania generowania pewnych składowych, np. funkcji, które nie są jeszcze w pełni opracowane lub powiązań z klasami, które nie są jeszcze w pełni zdefiniowane. Generator kodu powinien także generować i uaktualniać plik definiujący sposób kompilacji systemu, np. plik dla programu make w środowisku Unix.

Wygenerowany kod musi być z reguły uzupełniony poprzez dopisanie definicji procedur, funkcji i metod. Wygenerowany kod musi zawierać fragmenty, które mogą być modyfikowane przez użytkownika i pozostają nienaruszone przy powtórnym generowaniu kodu. Niektóre narzędzia, np. objectiF, pozwalają na wprowadzenie kodu metod i funkcji już na poziomie systemu CASE.

Narzędzia CASE nie narzucają zbyt wielu ograniczeń na stosowane w opisie projektu identyfikatory. Możliwe jest więc wprowadzanie długich nazw, zawierających spacje i znaki specjalne, w tym znaki polskie. Kompilatory z reguły nie dopuszczają tak swobodnej składni identyfikatorów. Generator kodu powinien więc w sensowny sposób, nie prowadzący do niejednoznaczności, skracać zbyt długie nazwy oraz usuwać/zastępować niepoprawne znaki. W przeciwnym wypadku już na etapie analizy niezbędne będzie posługiwanie się nazwami dostosowanymi do ograniczeń danego kompilatora. Może to w wyraźny sposób zmniejszyć czytelność projektu.

Moduł zarządzania wersjami

Moduł taki wspomaga zarządzanie projektami opisującymi różne wersje tego samego systemu (porównaj sekcję 13.6 omawiającą zarządzanie wersjami).

Moduł projektowania interfejsu użytkownika

Moduł ten służy do dialogowego projektowania składowych interfejsu użytkownika, na przykład dialogów, okien, menu. W stosunku do narzędzi tego typu, które nie wchodzą w skład narzędzi CASE, zaletą takich modułów jest wykorzystywanie informacji zawartej w słowniku danych. Pozwala to np. na automatyczne wygenerowanie dialogu służącego do edycji pewnej struktury danych (porównaj rysunek 6.16), który następnie musi jedynie zostać odpowiednio sformatowany. Moduł tego typu wchodzi np. w skład pakietu System Architect. Podobne możliwości można osiągnąć również przez integrację systemu CASE z narzędziami RAD.

Moduł(y) inżynierii odwrotnej

Są to moduły, których celem jest odtwarzanie zawartości słownika danych oraz, w niektórych wypadkach diagramów, na podstawie istniejącego kodu (czasami także na podstawie struktury istniejącej bazy danych).

Sprzęg(i) do narzędzi RAD

Wiele narzędzi CASE posiada moduły pozwalające na ścisłą współpracę z narzędziami RAD. W przypadku pakietu Oracle CASE są to np. narzędzia RAD firmy Oracle. Select OMT Professional współpracuje z systemem Visual Basic Enterprise Edition. System Architect może zostać zintegrowany z pakietem Power Builder. Visible Analyst Workbench zawiera sprzęgi do pakietów Power Builder, SQL Windows i kilku innych narzędzi RAD.

Sprzęgi do narzędzi RAD pozwalają na automatyczne generowanie składowych interfejsu użytkownika, definicji relacji oraz wiązanie elementów interfejsu użytkownika ze składowymi bazy danych na podstawie opisu projektu. Niektóre narzędzia pozwalają także na przepływ informacji w drugą stronę, to jest z pakietu RAD do pakietu CASE. Możliwe jest np. wprowadzenie definicji nowej encji na podstawie relacji zadeklarowanej w pakiecie RAD.

Moduł importu/eksportu danych

Wiele narzędzi CASE pozwala na wymianę danych z innymi pakietami tego typu. Opracowane zostały standardy wymiany danych, na przykład CDIF (CASE Data Interchange Format) opracowany przez Electronic Industries Association. Z reguły narzędzia CASE pozwalają także na eksport i import danych w prostej formie tekstowej.

Moduł zarządzania pracą grupową

Większość przedsięwzięć, w których wykorzystuje się narzędzia CASE wymaga współpracy wielu osób. Nad tym samym projektem może więc pracować wiele osób, które powinny mieć dostęp do fragmentów projektu. Moduł zarządzania pracą grupową powinien realizować następujące funkcje:

- dodawanie i usuwanie użytkowników oraz grup użytkowników mających prawo pracy nad projektem,

- ochronę dostępu do projektu za pomocą haseł,
- określanie praw użytkowników i grup użytkowników do odczytu i modyfikacji poszczególnych fragmentów projektu,
- udostępnianie przez uprawnionych użytkowników praw dostępu do tworzonych przez nich fragmentów projektu innym użytkownikom,
- zabezpieczanie fragmentów projektu przed przypadkową zmianą,
- śledzenie pracy poszczególnych użytkowników.

Moduł pracy sieciowej

W wielu pakietach CASE pracujących na komputerach klasy PC i stacjach roboczych możliwość równoczesnej pracy kilku osób nad tym samym projektem zapewnia umieszczenie plików opisujących projekt w ogólnie dostępnym katalogu sieciowym. Moduł pracy sieciowej powinien zapewniać blokowanie składowych edytowanych przez jednego z użytkowników. Blokowany powinien być oczywiście jak najmniejszy fragment projektu. Podobne możliwości pozwala osiągnąć zastosowanie wielodostępnej bazy danych do przechowywania słownika danych.

12.3. Narzędzia CASE w działalności firmy programistycznej

Narzędzia CASE, podobnie jak inne produkty, mają swój cykl życia z punktu widzenia użytkownika, którym w tym przypadku jest firma programistyczna. Na ten cykl życia składają się następujące fazy:

- wybór konkretnego systemu oraz jego konfiguracji,
- wdrażanie,
- użytkowanie i ewolucja,
- wycofanie z użytku.

Liczba dostępnych na rynku narzędzi CASE jest obecnie bardzo duża. Rynek tych narzędzi przeżywa obecnie etap gwałtownego rozwoju. Można się spodziewać, że wraz z upływem czasu kilku producentów osiągnie wyraźnie dominującą pozycję. Nabywca pakietu CASE musi więc brać pod uwagę ryzyko, że zakupiony produkt „wypadnie" z rynku i nie będzie dalej rozwijany. Należy jednak dodać, że w chwili pisania tej książki tylko część producentów narzędzi CASE ma poważnych dystrybutorów w Polsce.

Na ocenę narzędzi CASE wpływa szereg kryteriów takich, jak:

- *zakres oferowanych funkcji i ich zgodność z potrzebami firmy,*

- *koszt,*
- *niezawodność,*
- *opinia o producencie i dystrybutorze,*
- *dostępność na rynku pracy specjalistów znających dany pakiet.*

Poprzedni rozdział omawiający poszczególne składowe narzędzi CASE oraz ich podstawowe cechy może być pomocny podczas oceny zakresu oferowanych funkcji. Internet pozwala uzyskać wiele interesujących opinii na temat poszczególnych narzędzi.

Firma specjalizująca się w wytwarzaniu oprogramowania z wykorzystaniem konkretnego środowiska programistycznego, będzie poszukiwała pakietu CASE, który zapewnia ścisłą integrację z tym środowiskiem. Wielu producentów kompilatorów, systemów zarządzania bazami danych oraz narzędzi RAD zaleca stosowanie konkretnych narzędzi CASE. Niektórzy z nich są zresztą także producentami narzędzi CASE — na przykład firma Oracle.

Pakiety wielośrodowiskowe są szczególnie atrakcyjne dla firm, które wykonują oprogramowanie z wykorzystaniem różnych środowisk lub planują przejście na nowe środowisko implementacji. Narzędzia takie mogą również zredukować niekorzystne rezultaty uzależniania się firmy od jednego środowiska programistycznego.

Narzędzia wspomagające wiele metodyk analizy i projektowania są szczególnie atrakcyjne dla firm, które wykonują bardzo różnorodne oprogramowanie. Zastosowanie jednego narzędzia, które wspomaga szereg metod i notacji jest znacznie lepszym rozwiązaniem niż posługiwanie się kilkoma różnymi pakietami.

Do niedawna powszechnie wyrażano opinie, że obiektowe narzędzia CASE są zdecydowanie mniej zaawansowane niż narzędzia strukturalne. Narzędzia te rozwijano jednak bardzo intensywnie w ostatnich 2 – 3 latach. Wielu znanych producentów strukturalnych narzędzi CASE opracowało wersje obiektowe swoich pakietów. Do firm takich należą: SELECT Software Tools, Interactive Developmet Environments, microTOOL, Visible Systems i Popkin Software & Systems. Duże doświadczenie tych firm w produkcji oprogramowania CASE doprowadziło do powstania narzędzi, które w niczym nie ustępują najlepszym narzędziom strukturalnym.

Oceniając koszt pakietu CASE należy brać pod uwagę nie tylko jego cenę, lecz także:

- koszt szkoleń,
- koszt dostosowania sprzętu do wymagań pakietu.

Wdrażanie pakietu CASE obejmuje:

- skonfigurowanie pakietu stosownie do potrzeb i zgodnie ze standardami już stosowanymi w firmie,
- dostosowanie standardów firmy do nowej technologii,

- szkolenia pracowników w zakresie metodyki inżynierii oprogramowania, szczególnie analizy i projektowania,
- szkolenia pracowników w zakresie obsługi pakietu,
- odtworzenie dokumentacji technicznej poprzednich przedsięwzięć,
- wykonanie pilotowego projektu(ów) z wykorzystaniem pakietu CASE, często równolegle z tradycyjną realizacją tego samego systemu.

Konfigurowanie pakietu CASE polega między innymi na:

- skonfigurowaniu słownika danych tak, aby pozwalał na przechowywanie i edycję niezbędnych informacji,
- zdefiniowaniu niezbędnych raportów,
- zdefiniowaniu dokumentów, które będą generowane za pomocą generatora dokumentacji technicznej.

Odtworzenie dokumentacji technicznej poprzednich przedsięwzięć może polegać na:

- wykorzystaniu funkcji inżynierii odwrotnej,
- wykorzystaniu możliwości importu danych w jednym ze standardowych formatów.

Drugi z powyższych sposobów jest oczywiście możliwy wtedy, gdy firma korzystała już wcześniej z innego narzędzia CASE.

W trakcie użytkowania pakiet CASE podlega ewolucji związanej z dalszymi zmianami konfiguracji i zmianami wersji systemu.

Po pewnym czasie firma może zrezygnować z dalszego prowadzenia prac z wykorzystaniem danego pakietu. Przydatne na tym etapie okażą się możliwości eksportu danych, które pozwolą na przeniesienie istniejących projektów do nowego pakietu CASE.

Należy podkreślić, że narzędzia CASE nie są jakimś cudownym środkiem, który gwarantuje poprawę efektywności firmy programistycznej. Przeciwnie, istnieją badania, które wskazują, że stosowanie narzędzi CASE często przynosi znikome efekty. Z drugiej strony zetknąłem się z wieloma przykładami bardzo udanych zastosowań tego rodzaju narzędzi. Trudności związane ze stosowaniem narzędzi CASE wynikają z następujących czynników:

- *Traktowanie narzędzi CASE wyłącznie jako generatorów kodu.* Dla wielu osób narzędzia CASE to wyłącznie wysokiego poziomu narzędzia programowania pozwalające na tworzenie kodu w wygodny, graficzny i dialogowy sposób. Nie ma w tym wypadku mowy o poważnej analizie i projektowaniu systemu. Okazuje się jednak, że efekty związane z takim wykorzystaniem narzędzi są niewielkie gdyż:
  - Nakłady na implementację stanowią jedynie niewielki procent (około 15 – 30%) nakładów na opracowanie całego systemu. Ich redukcja nie zmniejsza więc w wyraźny sposób całkowitych kosztów.
  - Koszt błędów popełnianych w fazie implementacji jest stosunkowo niewielki. Zmniejszenie liczby błędów dzięki stosowaniu narzędzi CASE nie powoduje więc znaczących oszczędności.
  - Istnieją inne, często tańsze narzędzia, np. RAD, które pozwalają automatyzować fazę implementacji.
- *Nieznajomość metodyki analizy i projektowania.* Podobnie jak programy CAD, systemy CASE to narzędzia dla specjalistów. Ich stosowanie przez osoby nie znające metod wspomaganych przez narzędzia CASE nie przynosi sensownych rezultatów.
- *Niewłaściwa organizacja i zarządzanie przedsięwzięciem.* Nieuporządkowanie procesu produkcji oprogramowania, brak monitorowania postępów prac i kontroli jakości, oraz inne błędy kierownictwa mogą skutecznie zniwelować korzyści płynące ze stosowania narzędzi CASE.
- *Zbyt wysokie oczekiwania związane z wdrożeniem narzędzi CASE.* Nadmierne oczekiwania mogą się wiązać z następującymi czynnikami.
  - Ocenia się, że stosowanie narzędzi CASE może zredukować nakłady na wytwarzania oprogramowania o nie więcej niż 50%. Większe korzyści pojawiają się na etapie konserwacji, której koszty mogą zmniejszyć się kilkakrotnie. Duże korzyści pojawiają się także w momencie realizacji nowych przedsięwzięć podobnych do poprzednich. Narzędzia CASE pozwalają na łatwe wykorzystanie wyników, które mogą okazać się przydatne w nowym przedsięwzięciu.
  - Koszt wdrożenia pakietu CASE jest stosunkowo wysoki. Koszt ten może być łatwo niedoszacowany, jeżeli nie weźmie się pod uwagę np. kosztów szkoleń.
  - Efekty stosowania narzędzi CASE pojawiają się z pewnym opóźnieniem. Wspomniano już, że największe korzyści ze stosowania narzędzi CASE pojawiają się na etapie konserwacji. Należy wziąć pod uwagę również konieczność zdobycia przez pracowników firmy niezbędnego doświadczenia w stosowaniu metod analizy i projektowania. Z tego powodu nie zaleca się natychmiastowego wykonywania dużych przedsięwzięć z wykorzystaniem nowego pakietu CASE. Na początku należy podjąć probę realizacji stosunkowo niedużego przedsięwzięcia pilotowego.

Informacja o producentach wymienionych narzędzi

Producentem pakietu Oracle CASE jest firma Oracle.

Producentem pakietu Select OMT Professional jest firma SELECT Software Tools.

Producentem pakietów z serii Software through Pictures (StP) jest firma Interactive Developmet Environments.

Producentem pakietu Paradigm Plus jest firma ProtoSoft.

Producentem pakietów CASE 4.0 i objectiF jest firma microTOOL.

Producentem pakietu Visible Analyst Workbench jest firma Visible Systems.

Producentem pakietu EasyCASE jest firma Evergreen CASE Tools.

Producentem pakietu System Architect jest firma Popkin Software & Systems.

Rozdział 13. Zarządzanie przedsięwzięciem programistycznym

Zatrudnienie wysokiej klasy specjalistów dobrze znających metody inżynierii oprogramowania oraz stosowanie zaawansowanych narzędzi wspomagających stosowanie tych metod nie gwarantuje sukcesu przedsięwzięcia programistycznego. Niezbędnym warunkiem sukcesu jest właściwe zarządzanie przedsięwzięciem.

Podstawowe zadania kierownictwa przedsięwzięcia programistycznego to:

- opracowywanie propozycji dotyczących sposobu prowadzenia przedsięwzięcia,
- kosztorysowanie przedsięwzięcia,
- planowanie i harmonogramowanie przedsięwzięcia,
- monitorowanie i kontrolowanie realizacji przedsięwzięcia,
- dobór i ocena personelu,
- opracowywanie i prezentowanie sprawozdań dla kierownictwa wyższego szczebla.

Sposoby zarządzania przedsięwzięciem programistycznym są oczywiście wzorowane na sposobach zarządzania innego rodzaju przedsięwzięciami. Z drugiej strony muszą one brać pod uwagę specyfikę przedsięwzięć programistycznych, na przykład nieprzejrzystość procesu budowy oprogramowania i inne wymienione w sekcji 1.1, omawiającej kryzys oprogramowania oraz jego przyczyny.

13.1. Czynniki psychologiczne w inżynierii oprogramowania

Czynniki psychologiczne są ważne w inżynierii oprogramowania z dwóch powodów:

- oprogramowanie jest w większości wypadków użytkowane przez ludzi,
- oprogramowanie jest tworzone przez ludzi.

Fakt, że wiele systemów jest użytkowanych przez ludzi ma duży wpływ na zasady projektowania interfejsu użytkownika i tworzenia dokumentacji użytkowej. Są to więc zagadnienia, które były omówione w sekcji 6.2.1 omawiającej projektowanie składowej kontaktu z użytkownikiem i w rozdziale 8 omawiającym zagadnienia dokumentacji użytkowej. W dalszej części rozdziału poruszane będą zagadnienia psychologiczne odgrywające rolę na etapie budowy oprogramowania. Z zagadnień tych wypływają istotne praktyczne wnioski dla kierownictwa przedsięwzięcia programistycznego. Pozwalają też one w bardziej obiektywny sposób uzasadnić znaczenie, niektórych omówionych już wcześniej zasad inżynierii oprogramowania.

Ostatnie lata przyniosły poważne zmiany w rozumieniu istoty ludzkiej inteligencji. Zmiany te są między innymi wynikiem trwających już od wielu lat prac w zakresie sztucznej inteligencji. W początkowym okresie rozwoju tej dziedziny powszechnie zakładano, że ludzka inteligencja musi opierać na zasadach analogicznych do tych, które próbowano stosować w pierwszych systemach ekspertowych, to jest:

- na analitycznym podejściu do problemu, czyli rozbijaniu złożonego problemu na prostsze podproblemy,
- stosowaniu ścisłych reguł logicznego wnioskowania.

Pozwalało to wierzyć, że w krótkim czasie uda się zautomatyzować wszelkie zadania wymagające ludzkiej inteligencji, a więc także wytwarzanie oprogramowania.

Mimo niepodważalnych osiągnięć sztucznej inteligencji z czasem stało się jasne, że sposób pracy sztucznych systemów zdecydowanie różni się od sposobu pracy ludzkiego umysłu. Nadal istnieje szereg problemów, w przypadku których ludzie radzą sobie znacznie lepiej niż najlepsze komputery, lub komputery dorównują człowiekowi jedynie dzięki stosowaniu techniki brutalnej siły, czyli ogromnej liczby obliczeń. Zagadnienia te są w pasjonujący sposób omawiane w książce Dreyfusa i Dreyfusa (1986). Autorzy ci argumentują, że istotnymi elementami ludzkiej inteligencji są:

- umiejętność całościowego (syntetycznego) spojrzenia na problem,
- posługiwanie się wiedzą płynącą z doświadczenia, a więc stosowanie nieścisłych zasad wnioskowania na bazie wcześniejszych doświadczeń.

Powyższe umiejętności są trudne do naśladowania przez sztuczne systemy. Umiejętności te mają duże znaczenie w pracach nad oprogramowaniem, w szczególności we wstępnych fazach realizacji przedsięwzięcia, kiedy niezbędne jest całościowe ujęcie dużego systemu oraz wykorzystanie zdobytego wcześniej doświadczenia. Ma to niewątpliwy wpływ na trudności, jakie napotkały próby pełnego sformalizowania procesu wytwarzania oprogramowania (porównaj dyskusję na temat nurtu formalnego i praktycznego w inżynierii oprogramowania w sekcji 1.2).

Jest oczywiste, że pewne osoby lepiej niż inne radzą sobie w pracy nad budową oprogramowania. Wydajność poszczególnych osób może różnić się o więcej niż rząd wielkości. Z punktu widzenia kierownictwa firmy programistycznej idealnym rozwiązaniem byłaby możliwość zidentyfikowania szczególnie przydatnych osób już na etapie, w którym rozważane jest ich zatrudnienie. Wiele firm nieinformatycznych poszukujących pracowników posługuje się tzw. testami osobowości, których celem jest stwierdzenie czy dana osoba posiada cechy przydatne w pracy na rozważanym stanowisku. Podobnymi testami zaczynają posługiwać się także firmy programistyczne, w tym także niektóre firmy polskie. Testy osobowości z reguły mają formę zestawów pytań. Na podstawie udzielonych odpowiedzi specjalista psycholog określa cechy osobowości (tzw. profil osobowości) danej osoby. Ze stosowaniem tego typu testów wiążą się jednak następujące trudności:

- Osobowość ludzka ma charakter dynamiczny. Pożądane cechy osobowości inżyniera oprogramowania określane są na podstawie cech osób od wielu lat zajmujących się produkcją oprogramowania i osiągających w tej dziedzinie bardzo dobre wyniki. Jest jednak oczywiste, że wieloletnia praktyka zawodowa nie pozostaje bez wpływu na osobowość. Powstaje więc pytanie, na ile pożądane cechy osobowości doświadczonych inżynierów oprogramowania zostały przez nich nabyte podczas pracy w firmie programistycznej, a na ile są to cechy wrodzone.
- Różne zadania mogą wymagać innych cech osobowości. Należy się spodziewać, że inne cechy będą przydatne analitykowi (związane na przykład z koniecznością częstych kontaktów z klientem), inne projektantowi, programiście czy osobie zajmującej się testowaniem oprogramowania. Powinno się także wziąć pod uwagę zmiany zachodzące w inżynierii oprogramowania, które mogą prowadzić do zmian wymagań wobec osób pracujących w firmach programistycznych.
- Należy też pamiętać, że osoby odpowiadające na pytania testowe będą na pewno próbowały odgadnąć pożądane odpowiedzi. Wyniki testu niekoniecznie więc będą odzwierciedlać rzeczywiste cechy danej osoby, lecz raczej jej wyobrażenie o cechach pożądanych przez pracodawcę (można oczywiście stwierdzić, że umiejętność odgadywania, na przykład potrzeb klienta, jest pożądaną cechą inżyniera oprogramowania).

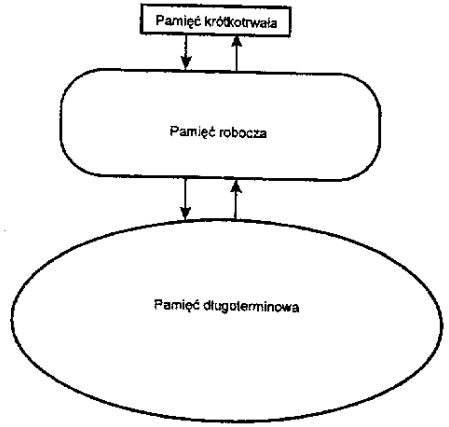
Liczba powszechnie akceptowanych cech dobrego inżyniera oprogramowania jest stosunkowo niewielka. Należą do nich:

- *Umiejętność pracy w stresie.* W pracy inżyniera oprogramowania często zdarzają się okresy wymagające szybkiego wykonania złożonych zadań. Dla większości osób niewielki stres ma działanie mobilizujące. Po przekroczeniu pewnego progu

następuje jednak spadek możliwości danej osoby. Poszczególne osoby różnią się jednak znacznie wielkością tego progu. Do pracy w firmach programistycznych najbardziej nadają się osoby o stosunkowo wysokim progu odporności na stres.

- *Zdolności adaptacyjne.* Informatyka jest niewątpliwie jedną z najszybciej zmieniających się dziedzin. Ocenia się, że zaledwie 7 – 9 miesięcy przynosi w informatyce zmiany porównywalne z tymi, które w innych dziedzinach przemysłu zachodzą w ciągu 5 – 7 lat. Inżynier oprogramowania nieustannie musi zdobywać nowe umiejętności, poznawać nowe metody, narzędzia i sposoby pracy.

Na pracę inżyniera oprogramowania, podobnie jak na wszelką działalność intelektualną, istotny wpływ ma organizacja ludzkiej pamięci (rysunek 13.1).



Rysunek 13.1. Organizacja pamięci ludzkiej

Przyjmuje się, że pamięć można podzielić na trzy rodzaje:

- *Pamięć krótkotrwałą.* W pamięci tej umieszczane są informacje właśnie otrzymane przez daną osobę (na przykład ostatnio przeczytane zdanie tej książki). W pamięci tej umieszczane są też informacje, o których myślimy w danej chwili. Pamięć ta ma stosunkowo niewielką pojemność. Wspomniana już psychologiczna reguła 7±2 (Miller, 1956), mająca zastosowanie na przykład przy projektowaniu interfejsu użytkownika (porównaj sekcję 6.2.1), wynika z pojemności tej pamięci. Wielkość pamięci krótkoterminowej ma duży wpływ nie tylko na nasze zdolności postrzegania nowych informacji, lecz także na możliwość pracy nad złożonymi problemami.

- *Pamięć roboczą.* Jest to pamięć o znacznie większej pojemności. Przechowuje ona informacje, o których nie myślimy w danej chwili, lecz będące łatwo dostępne. Nowe informacje, umieszczane najpierw w pamięci krótkotrwałej, przechodzą następnie do pamięci roboczej (jeżeli tak się nie stanie, informacje te zostaną

natychmiast zapomniane). W pamięci roboczej przechowywane są fakty, nad którym dana osoba pracuje. Fragmenty opisu problemu będące bezpośrednim przedmiotem pracy w danej chwili są przenoszone do pamięci krótkotrwałej.

- *Pamięć długoterminową.* Pamięć ta ma ogromną pojemność. Informacje w niej zawarte nie zawsze są jednak dostępne w każdej chwili. Często przypomnienie sobie pewnego faktu wymaga sporego wysiłku i zajmuje sporo czasu. Zapomniane informacje z reguły nie są wymazywane z pamięci długoterminowej, a jedynie stają się trudno dostępne. Nowe fakty powinny być kilkakrotnie przypomniane, aby zostały dobrze utrwalone w pamięci długoterminowej.

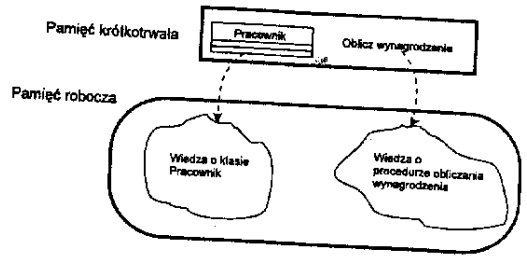
Przedstawiona powyżej organizacja ludzkiej pamięci przypomina organizację pamięci komputerowej. Pamięć krótkotrwała może odpowiadać szybkim rejestrom procesora, pamięć robocza — pamięci operacyjnej, a pamięć długoterminowa — pamięci masowej. Analogia ta nie jest jednak zupełna. Okazuje się, że pojemność ludzkiej pamięci należy mierzyć zupełnie inaczej niż pojemność pamięci komputerowej. Pamięć krótkotrwała mieści 5 – 9, lub raczej 3x3 porcje informacji. Czym jednak jest ta porcja? W przypadku zapamiętywania numerów telefonicznych, pojedynczą porcję można utożsamiać z jedną cyfrą. Większość osób rzeczywiście ma duże trudności z zapamiętaniem numerów dłuższych niż 7 cyfr. Warto też zauważyć, że zapisując lub wymawiając numer telefonu z reguły wprowadzamy wewnętrzne grupowanie cyfr. Większość osób uzna, że numer zapisany w następujący sposób:

735 63 34

jest czytelniejszy niż ten sam numer zapisany następująco:

7356334

W innych sytuacjach pojedyncza porcja informacji może oznaczać słowo, rysunek, lub skojarzenie do obszerniejszej informacji umieszczonej w pamięci roboczej. Możliwość umieszczania w pamięci krótkotrwałej skojarzeń do innych faktów ma istotny wpływ na możliwość pracy nad złożonymi problemami.



Rysunek 13.2. Pola skojarzeń w pracy nad złożonymi problemami

Rysunek klasy może być odwołaniem do całej wiedzy dotyczącej danej klasy, a nazwa procedury odwołaniem do obszernej wiedzy o sposobie jej realizacji. Posługiwanie się rysunkami i czytelnymi nazwami zdecydowanie ułatwia tworzenie tego typu skojarzeń.

jest to więc uzasadnienie roli jaką w inżynierii oprogramowania przypisuje się notacjom graficznym i zrozumiałemu nazewnictwu.

Pamięć ludzka, w szczególności pamięć długoterminowa, przechowuje informacje pochodzące ze świata zewnętrznego w mocno przetworzony sposób. Dla określenia informacji zawartych w pamięci używa się też terminu wiedza. Istnieją dwa główne rodzaje wiedzy:

- *Wiedza syntaktyczna* (*składniowa*). Do tego rodzaju można np. zaliczyć wiedzę na temat sposobu zapisu funkcji w języku C, lub wiedzę kiedy należy użyć operatora „:=", a kiedy operatora „=". Wiedza tego typu jest zapamiętywana w formie słabo przetworzonej. Wiedza ta jest zdobywana przez zapamiętywanie. Nowa wiedza tego typu jest w niewielkim stopniu integrowana ze zdobytą wcześniej, lecz jest raczej w prosty sposób dodawana do już istniejącej. Może to prowadzić do nakładania się wiedzy syntaktycznej dotyczącej np. różnych języków programowania. Doświadczył tego chyba każdy kto musiał w krótkich odstępach czasu posługiwać się różnymi, chociaż opartymi na podobnych koncepcjach, językami programowania — na przykład C i Pascalem.

- *Wiedza semantyczna* (*znaczeniowa*). Do tego rodzaju można zaliczyć np. rozumienie sposobu działania instrukcji *while*, znajomość koncepcji klasy w języku obiektowym, znajomość techniki szybkiego sortowania. Wiedza ta jest zdobywana przez doświadczenie i aktywne uczenie się. Nie może zostać po prostu zapamiętana. Przechowywana jest w formie mocno przetworzonej. Nowa wiedza tego typu jest integrowana z już istniejącą i rzadko się z nią nakłada. Jest na przykład bardzo mało prawdopodobne, aby osoba, która zrozumiała pojęcia klasy i procedury pomyliła je ze sobą w przyszłości.

Fakt istnienia powyższych dwóch rodzajów wiedzy może mieć istotny wpływ na politykę kadrową firmy programistycznej. Jeżeli firma poszukuje np. pracownika do pracy nad obiektowym systemem implementowanym w języku C++, to można się spodziewać, że bardziej przydatna będzie osoba znająca programowanie obiektowe w Pascalu niż osoba znająca podobny składniowo język C. Osoba doświadczona w programowaniu obiektowym stosunkowo łatwo zapozna się z inną składnią języka C++.

Wydaje się, że wiele firm zbyt dużą wagę przywiązuje do znajomości konkretnych środowisk i języków oprogramowania. W prasie można na przykład znaleźć wiele ogłoszeń, w których poszukiwani są pracownicy znający pewien konkretny język. Bardziej sensowne wydaje się stawianie wymagań związanych z posiadaniem pewnego rodzaju wiedzy semantycznej, na przykład znajomość systemów relacyjnych baz danych, programowania obiektowego, analizy obiektowej lub strukturalnej, programowania w środowiskach graficznych.

Oprogramowanie realizowane w firmach programistycznych jest z reguły na tyle duże, że nie może być wykonywane przez jedną osobę. Praca inżynierów oprogramowania jest więc najczęściej pracą zespołową. Pewne badania wskazują, że inżynier oprogramowania spędza przeciętnie połowę swojego czasu w bezpośrednim kontakcie z innymi

członkami zespołu. Czynniki psychologiczne związane z pracą grupową są więc bardzo istotne z punktu widzenia efektywności całego zespołu. Z punktu widzenia nastawienia do pracy grupowej można wyróżnić następujące typy osób:

- *Zorientowani na zadanie* (ang. *task-oriented*). Są to osoby, które można określić jako: samowystarczalne, zdolne, introwertyczne, agresywne, lubiące współzawodnictwo, niezależne.

- *Zorientowani na siebie* (ang. *self-oriented*). Są to osoby, które można określić jako: niezgodne, dogmatyczne, lubiące współzawodnictwo, agresywne, introwertyczne, zazdrosne.

- *Zorientowani na interakcję* (ang. *interaction-oriented*). Są to osoby, które można określić jako: nieagresywne, posiadające niewielką potrzebę autonomii i osiągnięć, pomocne, przyjazne.

Osoby zorientowane na zadanie są bardzo wydajne, jeżeli pracują indywidualnie. Okazuje się jednak, że zespoły złożone wyłącznie z takich osób osiągają stosunkowo słabe wyniki. Kilka indywidualności może nie być w stanie stworzyć prawdziwego zespołu. Dobry zespół powinien więc obejmować zarówno osoby zorientowane na interakcję, jak i zorientowane na zadanie. Osoby zorientowane na siebie także mogą być bardzo dobrymi członkami zespołu, jeżeli są przez kierownictwo przedsięwzięcia w odpowiedni sposób motywowane. Z drugiej strony osoby zorientowane na zadanie mogą stać się osobami zorientowanymi na siebie, jeżeli ich praca nie jest w odpowiedni sposób doceniana. Osoby zorientowane na interakcję okazują się również szczególnie przydatne we wstępnych fazach przedsięwzięcia wymagających intensywnych kontaktów z przedstawicielami klienta.

Z pracą grupową związana jest idea wspólnego programowania (ang. *egoless programming*). Ten sposób pracy zakłada, że realizacja postawionych przed grupą obowiązków jest wspólnym zadaniem całego zespołu, a wyniki pracy grupy są wspólną „własnością" wszystkich jej członków. Praca grupy jest więc traktowana jako wspólny wysiłek, a nie sumę indywidualnych działań.

Wspólne programowanie może jednak prowadzić do pojawienia się tzw. lojalności grupowej. Pod tym pojęciem rozumie się występowanie silnego, wręcz osobistego związku pomiędzy członkami zespołu a całością grupy i wynikami jej pracy. Może to prowadzić do następujących niekorzystnych efektów:

- *Trudność zmiany lidera.* Silnie związana grupa może nie zaakceptować lidera pochodzącego z zewnątrz. Może się okazać, że formalny lider nie będzie zdolny do rzeczywistego kierowania grupą, a rolę faktycznego lidera będzie pełnić inna osoba. Utrudnia to oczywiście zdecydowanie zarządzane tym zespołem.

- *Myślenie grupowe* (ang. *groupthink*). Pojęcie to oznacza brak krytycznego spojrzenia na efekty pracy grupy oraz nie rozważanie jakichkolwiek rozwiązań poza przyjętymi przez grupę. Poszczególne osoby, nawet z reguły krytycznie nastawione wobec swoich indywidualnych możliwości, zaczynają uważać grupę za nieomylną. Rezultatem jest oczywiście znaczny spadek jakości pracy.

Myślenia grupowego można unikać poprzez:

- *Sesje krytyki*, czyli specjalne zebrania, podczas których dozwolona jest jedynie krytyka przyjętych rozwiązań i efektów pracy. Zabroniona jest natomiast jakakolwiek obrona osiągnięć grupy.
- Włączanie do zespołu *krytycznych osobowości* — osób o szczególnych skłonnościach do wyszukiwania błędów i kwestionowania przyjętych rozwiązań. Osoby takie mogą się oczywiście spodziewać, że nie będą lubiane w grupie, co zresztą często nie stanowi dla nich z reguły istotnego problemu.

Na efekty pracy inżynierów oprogramowania wpływa także ergonomia miejsca pracy. W pewnym okresie, szczególnie w Stanach Zjednoczonych, pomieszczenia firm programistycznych urządzano w stylu amerykańskich biur. Były to więc duże sale, zawierające kilkanaście lub kilkadziesiąt stanowisk pracy. Obecnie uważa się, że lepsze wyniki zapewnia umieszczenie dwóch lub trzech stanowisk pracy w mniejszych pomieszczeniach. Poszczególne osoby powinny mieć także możliwość personalizacji swoich stanowisk pracy.

Pomieszczenia firmy programistycznej powinny też obejmować pokój zebrań służący do organizacji formalnych spotkań pracowników. Miejscem nieformalnych spotkań powinno być wspólne pomieszczenie komunalne, przystosowane np. do przygotowywania kawy. Praktyka pokazuje, że rozwiązania poważnych problemów pojawiają się często podczas nieformalnych spotkań, a nie podczas formalnych zebrań.

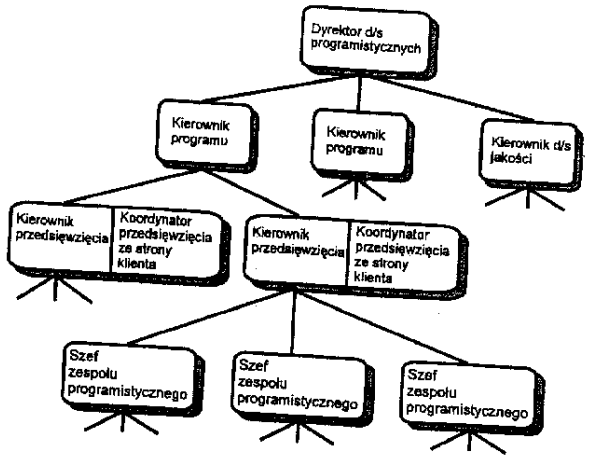
Na ergonomię pracy inżyniera oprogramowania wpływa także wykorzystywany sprzęt komputerowy. Większość osób czuje się znacznie bardziej komfortowo i osiąga lepsze wyniki, jeżeli pracuje na sprzęcie najnowszej generacji i w rozbudowanej konfiguracji, także wtedy, gdy trudno jest uzasadnić to w merytoryczny sposób.

Na zakończenie chciałbym podkreślić, że czynniki psychologiczne nie są jedynym, ani nawet głównym, elementem wpływającym na efekty współpracy inżynierów oprogramowania. Podstawowym warunkiem dobrej współpracy jest sprawna komunikacja pomiędzy poszczególnymi osobami. Jak wspomniano w sekcji 5.1, omawiającej rodzaje i role notacji wykorzystywanych w fazie analizy, podstawowym warunkiem sprawnej komunikacji jest posługiwanie się notacjami pozwalającymi na szybkie i precyzyjne przekazywanie informacji. O współpracy pomiędzy inżynierami oprogramowania można zresztą mówić także wtedy, gdy nie kontaktują się oni osobiście. Pracownik pracujący nad modyfikacją oprogramowania w fazie konserwacji, współpracuje np. z wykonawcami systemu, posługując się stworzonym przez nich projektem. Komfort psychiczny nie poparty dobrym przygotowaniem technicznym, nie zapewni więc dobrych wyników pracy grupy inżynierów oprogramowania.

13.2. Struktura zarządzania firmą programistyczną/działem programistycznym

Rysunek 13.3 przedstawia typową strukturę zarządzania firmą programistyczną lub działem programistycznym przedsiębiorstwa, którego główną działalnością nie jest produkcja oprogramowania. Prezentowana struktura jest stosunkowo rozbudowana, dotyczy więc dość dużych firm, realizujących jednocześnie wiele przedsięwzięć. Kierownicy programów są odpowiedzialni za koordynowanie prac kilku, podobnych przedsięwzięć. Podobieństwo to może wynikać ze stosowania podobnych narzędzi i metod i/lub z podobnego przeznaczenia oprogramowania. Kierownik przedsięwzięcia jest osobą odpowiedzialną za zarządzanie pojedynczym przedsięwzięciem programistycznym. Jego partnerem powinien być koordynator przedsięwzięcia ze strony klienta. Przedsięwzięcie programistyczne jest realizowane nie tylko przez pracowników firmy programistycznej, lecz także przez szereg osób ze strony zleceniodawcy. Kierownik przedsięwzięcia i koordynator przedsięwzięcia ze strony klienta są wspólnie odpowiedzialni za właściwą współpracę podległych im osób.

Na podkreślenie zasługuje wysoka pozycja kierownika d/s jakości. Powinien on podlegać bezpośrednio dyrektorowi d/s programistycznych, a więc być w pełni niezależnym od kierowników przedsięwzięć i programów.



Rysunek 13.3. Struktura zarządzania firmą programistyczną/działem programistycznym

Zespołu programistycznego (ang. *programming team*) nie należy utożsamiać z zespołem programistów. Pod tym pojęciem rozumie się także zespoły inżynierów oprogramowania pełniących funkcje analityków lub projektantów.

Osoba pracująca nad oprogramowaniem może pełnić następujące funkcje:

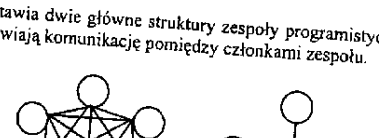
- *Kierownik programu/przedsięwzięcia.*
- *Analityk* — osoba kontaktująca się bezpośrednio z klientem, której celem jest określenie jego wymagań i budowa modelu systemu.
- *Projektant* — osoba odpowiedzialna za realizację oprogramowania. W zależności od zakresu prac można tu wyróżnić następujące bardziej szczegółowe funkcje:
  - *Projektant interfejsu użytkownika* — osoba odpowiedzialna za zaprojektowanie ergonomicznego i zgodnego ze standardami firmy interfejsu użytkownika.
  - *Projektant bazy danych* — osoba odpowiedzialna za zaprojektowanie i dostrojenie bazy danych.
- *Programista* — osoba implementująca oprogramowanie.
- *Osoba wykonująca testy.*
- *Osoba odpowiedzialna za konserwację oprogramowania.*
- *Ekspert metodyczny* — osoba szczególnie dobrze znająca stosowaną metodykę.
- *Ekspert techniczny* — osoba szczególnie dobrze znająca obsługę narzędzi.

W dużych przedsięwzięciach poszczególne osoby mogą pełnić tylko jedną z powyższych funkcji. W mniejszych przedsięwzięciach jedna osoba pełni wiele funkcji. W szczególności należy rozważyć sposób łączenia podstawowych funkcji: analityka, projektanta oraz programisty. W przypadku bardzo małych przedsięwzięć poszczególne osoby pełnią wszystkie te funkcje. W przypadku większych przedsięwzięć stosuje się następujące podejścia:

- Wyróżniane są dwie funkcje: analityka/projektanta i programisty. Osoba kontaktująca się z klientem opracowuje więc nie tylko wymagania klienta i model systemu, lecz również przygotowuje projekt oprogramowania. Od programistów nie wymaga się wtedy zbyt wysokiego poziomu zaawansowania. Mogą to więc być osoby, które ukończyły podstawowe kursy w zakresie programowania lub uzyskały wykształcenie średnie o profilu informatycznym. Należy jednak zaznaczyć, że polski system edukacji informatycznej nie jest ukierunkowany na kształcenie programistów nie posiadających wyższego wykształcenia. Firmy przyjmujące ten model mają więc trudności ze znalezieniem programistów. Praca programisty często nie zaspokaja ambicji osób z wyższym wykształceniem informatycznym. Osoby te stosunkowo szybko rezygnują więc z takiej pracy.
- Wyróżniane są funkcje analityka i projektanta/programisty. Osoba kontaktująca się z klientem jest odpowiedzialna za określenie wymagań, konstrukcję modelu i, co najwyżej, za wykonanie projektu wysokiego poziomu. Inne osoby są odpowiedzialne za opracowanie szczegółowego projektu i implementację oprogramowania. W tym wypadku osoby pełniące rolę projektanta/programisty muszą być znacznie bardziej zaawansowane.

Rysunek 13.4 przedstawia dwie główne struktury zespoły programistycznego. Linie na tym rysunku przedstawiają komunikację pomiędzy członkami zespołu.

Rysunek 13.4. *Sieciowa i gwiaździsta struktura zespołu programistycznego*



W zespole o strukturze sieciowej każdy z jego członków komunikuje się i współpracuje z pozostałymi. Zalety takiej organizacji to:

- Dzięki ścisłej współpracy członkowie zespołu wzajemnie kontrolują swoją pracę. Szybko wykrywane są więc odstępstwa od przyjętych standardów jakości.
- Struktura sieciowa umożliwia realizację idei wspólnego programowania (sekcja 13.1 omawia czynniki psychologiczne w inżynierii oprogramowania).
- Ponieważ praca poszczególnych osób jest dobrze znana innym członkom zespołu, stosunkowo łatwo mogą one przejąć obowiązki pracownika, który go opuścił.

Zespoły tego typu nie mogą być zbyt duże. Nie powinny liczyć więcej niż osiem osób. Ponadto wszyscy członkowie zespołu powinni posiadać podobne doświadczenie i podobny poziom zaawansowania. Struktura taka nie sprawdza się, jeżeli w skład zespołu wchodzą zarówno osoby początkujące, jak i osoby z dużym doświadczeniem.

W zespole o strukturze gwiaździstej szef zespołu jest jedyną osobą ściśle współpracują z pozostałymi osobami. Przydziela on zadania poszczególnym osobom i kontroluje efekty ich pracy. Jeżeli zachodzi konieczność wymiany informacji pomiędzy członkami zespołu, powinna się ona odbywać wyłącznie za pośrednictwem szefa zespołu.

Struktura gwiaździsta jest szczególnie przydatna, jeżeli w skład zespołu wchodzi wielu niedoświadczonych pracowników. Wielkość takiego zespołu może być większa niż zespołu o strukturze sieciowej. Liczba członków zespołu jest ograniczona przez możliwości kierowania pracą poszczególnych osób przez szefa zespołu. Wadą struktury gwiaździstej są duże problemy pojawiające się w momencie odejścia szefa zespołu. W tym wypadku jest to jedyna osoba znająca pracę całości zespołu. Pozostali członkowie znają pracę zespołu tylko fragmentarycznie.

W wielu firmach programistycznych typową drogą awansu jest przejście do struktur zarządzania. Praca związana z zarządzaniem cieszy się często większym prestiżem i jest lepiej płatna niż praca analityków, projektantów i programistów. Prowadzi to do tego, że najlepsi specjaliści odchodzą od pracy nad bezpośrednią realizacja oprogramowania.

Kierownictwo firmy powinno wypracować sposoby nagradzania i motywowania najlepszych specjalistów bez przenoszenia ich na stanowiska kierownicze.

13.3. Zapewnianie jakości

Zapewnianie i kontrola jakości bywają mylone z testowaniem oprogramowania. Pojęcia te powinny być jednak rozumiane znacznie szerzej niż wykrywanie błędów w oprogramowaniu. Zapewnianie jakości można zdefiniować w następujący sposób (porównaj Bersoff, 1984):

> Zapewnianie jakości to zestaw procedur, technik i narzędzi mających zapewnić, że tworzony produkt spełnia narzucone standardy. Jeśli standardy nie są jawnie określone, zapewnianie jakości dąży do zaspokojenia minimalnych, rynkowych wymagań co jakości.

Jakość oprogramowania należy rozumieć bardzo szeroko. Nie chodzi tu więc tylko o niezawodność, lecz także o pozostałe, wymienione już w rozdziale 1.2, omawiającym zakres inżynierii oprogramowania, kryteria jakości:

- zgodność z wymaganiami użytkownika,
- efektywność,
- łatwość konserwacji,
- ergonomiczność.

Na jakość oprogramowania wpływają działania podejmowane we wszystkich fazach jego życia. Może to prowadzić do nieporozumień. Zgodnie z definicja podaną w sekcji 1.2, celem całej inżynierii oprogramowania jest właśnie dążenie do wysokiej jakości oprogramowania. Istotnie, w wielu firmach zapewnianie jakości, nie jest oddzielane od pozostałych działań podejmowanych w trakcie trwania przedsięwzięcia programistycznego. Do czynności związanych z zapewnianiem jakości nie powinno się jednak zaliczać działań o charakterze typowo produkcyjnym.

Znaczenie poszczególnych kryteriów jakości może się znacznie różnić w zależności od charakteru danego przedsięwzięcia. Jednym z najczęściej popełnianych błędów jest założenie, że jakość jest powszechnie rozumiana w jednoznaczny sposób. Dlatego też, już na wstępie przedsięwzięcia, powinny zostać zdefiniowane kryteria jakości brane pod uwagę w danym przypadku. Kryteria te powinny być zawarte w dokumencie nazywanym planem jakości.

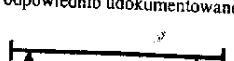
Zapewnianie jakości opiera się na założeniu, że jakość produktu jest silnie związana z jakością procesu, w trakcie którego powstaje produkt. Na tym założeniu opierają się między innymi coraz powszechniej wykorzystywane normy ISO 900x. Założenie to jest niezwykle istotne, gdyż ocena jakości procesu jest w ogólności łatwiejsza niż ocena jakości produktu, ponadto może być realizowana na długo przed powstaniem ostatecznego produktu.

Zapewnianie jakości polega więc między innymi na definiowaniu standardów procesu wytwarzania oprogramowania tak, aby zapewniał on zawsze osiągnięcie produktu wysokiej jakości. Definiując takie standardy dana firma może posłużyć się już istniejącymi standardami lub tworzyć własne. Te dwa podejścia nie wykluczają się oczywiście wzajemnie, gdyż stworzone na zewnątrz standardy nigdy nie będą w pełni odpowiadały potrzebom danej firmy. Należy dodać, że standaryzacja procesów w wytwarzania oprogramowania jest mniej zaawansowana niż ma to miejsce w innych dziedzinach techniki.

Standardy procesu nie są rzecz jasna celem samym w sobie. Muszą one podlegać weryfikacji, poprzez ocenę jakości wytworzonych produktów. Rozwiązania przyjęte „ad hoc" w pewnym przedsięwzięciu, mogą zostać uznane za standardy, jeżeli doprowadziły do powstania produktu o wysokiej jakości.

Jak wspomniano w sekcji 13.2 omawiającej strukturę zarządzania firmą programistyczną, dział zarządzania jakości powinien być wydzielonym pionem w firmie i podlegać bezpośrednio kierownictwu wyższego szczebla. Nie oznacza to, że standardy powinny powstawać bez udziału osób bezpośrednia zaangażowanych w produkcję oprogramowania. Przeciwnie, podkreśla się, że najlepsze standardy są tworzone przez bardzo zajętych inżynierów dobrze znających potrzeby produkcji. Dział zapewniania jakości jest odpowiedzialny między innymi za weryfikację poprawności i spójności standardów, ich redakcję, rozpowszechnianie i kontrolowanie stosowania.

Zapewnianie jakości wymaga oczywiście weryfikacji, czy proces jest realizowany zgodnie ze zdefiniowanymi standardami. Kontrole jakości dotyczą poszczególnych zadań. Mogą to być zadania bardzo ogólne, np. projektowanie, implementacja, lub też zadania znacznie bardziej szczegółowe, np. projektowanie konkretnego fragmentu systemu. Najlepiej jest, jeżeli w trakcie realizacji zadania przeprowadza się dwie kontrole (porównaj rysunek 13.5). Wstępna kontrola powinna być wykonana wkrótce po rozpoczęciu realizacji zadania. Jej celem jest sprawdzenie, czy już na wstępie nie popełnione zostały poważne błędy, np. czy stosowane są właściwe notacje i narzędzia. Końcowa kontrola powinna być wykonana na krótko przed zakończeniem realizacji zadania. W razie wykrycia pewnych błędów pozostawia to jeszcze pewien czas na ich usunięcie. W trakcie tej kontroli sprawdza się np. czy wykonano wszystkie niezbędne czynności i czy wyniki pracy są odpowiednio udokumentowane.



Rysunek 13.5. Momenty kontroli jakości w trakcie realizacji zadania

13.4. Poziomy rozwoju firmy programistycznej

Jak wspomniano w poprzednim rozdziale, jakość końcowego produktu jest ściśle związana za jakością procesu, w trakcie którego produkt ten powstaje. Oceniać poziom roz-

woju organizacji (firmy programistycznej), można więc na podstawie oceny poziomu procesu jaki wykorzystuje ta organizacja. Wyróżnić można pięć poziomów rozwoju organizacji z punktu widzenia dojrzałości procesu:

- *Początkowy* (ang. *initial*). Na tym poziomie nie istnieją żadne standardy procesu. Na poszczególnych etapach przedsięwzięcia decyzje dotyczące kolejności oraz sposobów realizacji dalszych działań podejmowane są „ad hoc". Faktu, że firma znajduje się na tym poziomie, nie należy utożsamiać z niskim poziomem zaawansowania technicznego. Możliwe jest więc, że firma znajdująca się na poziomie początkowym zatrudnia bardzo dobrych specjalistów i posługuje się zaawansowanymi narzędziami.

- *Powtarzalny* (ang. *repeateable*). Poszczególne przedsięwzięcia wykonywane są w podobny sposób. W firmie istnieje powszechna zgoda co do sposobu prowadzenia przedsięwzięć. Można więc mówić o pewnych standardach, które są jednak standardami „de facto". Standardy te nie są udokumentowane. Nie istnieją też ścisłe procedury kontroli, pozwalające zweryfikować, czy proces przebiega zgodnie ze standardami. Sukces przedsięwzięcia zależy więc mocno od indywidualnego doświadczenia oraz umiejętności poszczególnych osób zarządzających przedsięwzięciami.

- *Zarządzany* (ang. *managed*). Standardy procesu są dobrze zdefiniowane i sformalizowane. Zgodność poszczególnych przedsięwzięć ze standardami podlega ścisłej kontroli. W firmie istnieje wyróżniona grupa osób odpowiedzialnych za opracowywanie i uaktualnianie standardów oraz za kontrole zgodności przedsięwzięć z tymi standardami.

- *Mierzony* (ang. *measured*). Proces podlega nie tylko kontrolom weryfikującym zgodność ze standardami, lecz jest także „mierzony" w sposób ilościowy. Obserwacji podlegają pewne mierzalne charakterystyki procesu, np. wydajność. Wyniki tych pomiarów są przechowywane i wykorzystywane dla poprawy sposobu realizacji przyszłych przedsięwzięć.

- *Optymalizowany* (ang. *optimized*). Standardy są w ciągły sposób uaktualniane tak, aby biorąc pod uwagę doświadczenia wyniesione z poprzednich przedsięwzięć, zapewnić jak najlepsze efekty przyszłych działań. Standardy obejmują też elementy pozwalające na „dostrojenie" procesu do potrzeb danego przedsięwzięcia, zapewniające osiągnięcie produktu o wysokiej jakości przy rozsądnych kosztach. Na poziomie tym istotną rolę odgrywają obserwacje ilościowe.

Firma powinna oczywiście dążyć do osiągnięcia jak najwyższego poziomu. Firma znajdująca się na poziomie początkowym, powinna np. dążyć do uporządkowania procesu poprzez wprowadzenie powtarzalności podejmowanych działań. Firma znajdująca się na poziomie powtarzalnym, powinna udokumentować stosowane standardy, wprowadzić ścisłe kontrole procesu, oraz wyróżnić osoby odpowiedzialne za definiowanie i kontrolowanie procesu.

W odróżnieniu od innych dziedzin techniki, wydaje się, że pełne osiągnięcie poziomu zarządzanego, a tym bardziej optymalizowanego, przez firmę programistyczną nie jest jeszcze obecnie możliwe. Inżynieria oprogramowania nie jest jeszcze na tyle rozwinięta,

aby pozwalała na pełen ilościowy opis procesu produkcji oprogramowania. W sekcji 3.3 omawiającej algorytmiczne modele szacowania kosztów oprogramowania podano przykład stosowania metod ilościowych. Wspomniano też o konieczności posługiwania się danymi ilościowymi pochodzącymi z poprzednich przedsięwzięć w celu dostrojenia modelu algorytmicznego. W kolejnych rozdziałach zostaną omówione zagadnienia miar produktywności i harmonogramowania przedsięwzięć, niezbędne firmom pragnącym osiągnąć poziom mierzony lub optymalizowany.

Wiele pozycji dotyczących inżynierii oprogramowania sugeruje konieczność dążenia do jak najszerszego posługiwania się miarami ilościowymi w firmach programistycznych. Moim zdaniem trudność ujęcia ilościowego jest nieodłączną cechą oprogramowania. Dlatego niezbędne wydaje się stosowanie technik jakościowych oraz technik łączących elementy podejść ilościowych i jakościowych. W firmach dążących do osiągnięcia poziomu optymalizowanego dużą rolę powinny więc odgrywać techniki sztucznej inteligencji, automatycznego uczenia się, wnioskowania przybliżonego i wspomagania decyzji (porównaj np. Hapke i in., 1994). Są to jednak obecnie zagadnienia wymagające intensywnych badań i trudno tu o podanie zweryfikowanych w praktyce zaleceń. Dlatego też zagadnienia te wychodzą poza zakres niniejszej książki.

13.5. Rola dokumentacji w zarządzaniu przedsięwzięciem

W rozdziale 8 zostały omówione zagadnienia związane z dokumentacja użytkową, która stanowi część końcowego produktu. W trakcie prowadzenia przedsięwzięcia powstaje jednak szereg dokumentów przeznaczonych wyłącznie dla producenta oprogramowania. Dokumentacje taką można podzielić na:

- *dokumentację procesu* obejmującą dokumenty dotyczące samego procesu produkcji oprogramowania,

- *dokumentację techniczną* opisującą wytworzony produkt.

Oba powyższe rodzaje dokumentów mogą występować zarówno w formie drukowanej, jak i formie elektronicznej.

13.5.1. Dokumentacja procesu

Dokumentacja procesu obejmuje następujące rodzaje dokumentów:

- *Plany, szacunki, harmonogramy* — dokumenty tworzone przez kierownictwo przedsięwzięcia. Odbiorcami tych dokumentów mogą być przełożeni wyższego szczebla. W tym wypadku dokumenty te pełnią rolę propozycji pewnych działań. Zaakceptowane dokumenty tego typu przeznaczone są dla wykonawców przedsięwzięcia i pełnią rolę poleceń.

- *Raporty* — dokumenty przygotowywane przez kierowników niższego szczebla dla swoich przełożonych Są to sprawozdania opisujące przebieg dotychczasowych prac i uzyskane rezultaty.
- *Standardy* — dokumenty opisujące pożądany sposób realizacji przedsięwzięcia (porównaj sekcję 13.3. omawiającą zapewnianie jakości).
- *Dokumenty robocze* (ang. *working papers*) — rozmaite dokumenty zawierające wstępne propozycje pewnych rozwiązań. Twórcami i odbiorcami takich dokumentów są rozmaici członkowie zespołu realizującego przedsięwzięcie. Dokumenty robocze mogą zostać odrzucone. Zaakceptowane dokumenty tego typu zaczynają pełnić rolę standardów, raportów, planów lub dokumentacji technicznej.
- *Komunikaty* — rozmaite, z reguły krótkie dokumenty, służące do wymiany bieżących informacji pomiędzy członkami zespołu.

W przeszłości zakładano, że dokumentacja procesu, z wyjątkiem standardów, ma krótki cykl życia. Uważano, że dokumenty te stają się zbędne po zakończeniu przedsięwzięcia. Obecnie uważa się, że wykorzystanie dokumentacji procesu ma istotny wpływ na możliwość osiągnięcia przez firmę wyższych poziomów dojrzałości procesu. Dotyczy to szczególnie planów, szacunków i harmonogramów oraz raportów. Informacje w nich zawarte mogą dostarczyć szereg istotnych informacji o charakterze jakościowym i ilościowym pozwalających ocenić sposób prowadzenia poprzednich przedsięwzięć oraz trafność przewidywań kierownictwa. Informacje te powinny być wykorzystywane podczas opracowywania standardów dla przyszłych przedsięwzięć.

13.5.2. Dokumentacja techniczna

Dokumentacja techniczna jest opisem oprogramowania, przeznaczonym z reguły wyłącznie dla producenta oprogramowania. W pewnych przypadkach część dokumentacji technicznej może być jednak przekazywana nabywcy oprogramowania, zgodnie z zawartą umową.

Jak wspomniano już w sekcji 5.1. omawiającej rodzaje i role notacji wykorzystywanych w fazie analizy, dokumentacja techniczna powstaje podczas realizacji poszczególnych faz przedsięwzięcia. Nie jest więc wykonywana po zakończeniu implementacji i testów. Dokumentacja techniczna obejmuje wyniki poszczególnych faz, począwszy do fazy strategicznej aż do fazy testowania. Dokumentacja techniczna jest oczywiście nadal uaktualniana w fazie konserwacji. Poszczególne składowe dokumentacji technicznej zostały już wymienione w rozdziałach omawiających wyniki poszczególnych faz. Typowa dokumentacja techniczna obejmuje więc następujące składowe:

- wprowadzenie opisującego cele, zakres i kontekst systemu,
- opis ewolucji systemu, to znaczy opisu przewidywanych zmian wymagań wobec systemu,
- opis wymagań funkcjonalnych,

- opis wymagań niefunkcjonalnych.
- specyfikację wymagań funkcjonalnych (opcjonalnie),
- specyfikację wymagań niefunkcjonalnych (opcjonalnie),
- opis wymagań sprzętowych.
- opis wymagań dotyczących bazy danych,
- opis projektu, który w przypadku stosowania metod obiektowych składa się z:
  - diagramu(ów) klas,
  - diagramów interakcji obiektów,
  - diagramów przejść stanów,
  - innych diagramów o charakterze projektowym, np. diagramów Boocha,
  - raportów:
    - definicji klas,
    - definicji pól,
    - definicji danych złożonych oraz elementarnych,
    - definicji metod,

  a w przypadku stosowania metod strukturalnych z:
  - diagramu(ów) związków encji,
  - diagramów przepływów danych,
  - diagramów przejść stanów,
  - diagramów strukturalnych,
  - raportów:
    - definicji encji,
    - definicji atrybutów,
    - definicji procesów,
    - definicji zbiorników danych,
    - definicji przepływów danych,
    - definicji danych złożonych oraz elementarnych,
    - definicji modułów,
- raport przebiegu testów i oszacowanie niezawodności oprogramowania,
- kod źródłowy.

Przed przekazaniem oprogramowania do eksploatacji, dokumentacja techniczna może oczywiście zostać poddana weryfikacji, której celem jest wyeliminowanie błędów i niescisłości oraz zapewnienie zgodności z przyjętymi w firmie standardami. Procedury zapewniania jakości (porównaj sekcję 13.3 omawiającą te zagadnienia) powinny jednak zminimalizować konieczność wprowadzania zmian w dokumentacji po wykonaniu oprogramowania.

W firmie programistycznej, realizującej wiele przedsięwzięć ilość dokumentacji technicznej może być bardzo duża. Istotnym zagadnieniem jest więc właściwe zarządzanie dokumentacją techniczną. Wymaga to wdrożenia standardów dotyczących:

- procesu wytwarzania dokumentacji technicznej,
- treści i formy dokumentów,
- sposobu dostępu do dokumentacji.

Standaryzacja procesu wytwarzania dokumentacji wymaga określenia sposobu:

- tworzenia wstępnej wersji dokumentów,
- wygładzania — formatowania, korekcji błędów,
- produkcji — drukowania, powielania, oprawiania,
- wprowadzania zmian w istniejących już dokumentach i uaktualniania wersji dokumentów.

Niezbędne jest również ścisłe określenie osób odpowiedzialnych za wykonanie powyższych czynności. Dostępne w wielu narzędziach CASE generatory dokumentacji technicznej zdecydowanie ułatwiają tworzenie wstępnych wersji dokumentów i ich odpowiednie formatowanie.

Standaryzacja dokumentów polega na określeniu ich zawartości i formy. Typowa zawartość dokumentacji technicznej została wymieniona powyżej. Typowa struktura dokumentu obejmuje:

- stronę tytułową pomagającą w identyfikacji dokumentu,
- spis treści,
- rozdziały, podrozdziały i sekcje zawierające podstawową treść dokumentu,
- indeks,
- słownik.

Strona tytułowa może przykładowo zawierać następujące dane:

- nazwa — identyfikator przedsięwzięcia, którego dotyczy dokument,
- identyfikator wersji systemu,
- nazwa dokumentu, niezbędna jeżeli dokumentacja danego przedsięwzięcia jest dzielona na szereg dokumentów,
- autor (autorzy),
- data powstania,

- informacje dotyczące kontroli jakości — data zatwierdzenia, osoba zatwierdzająca dokument,
- informacje dotyczące stopnia tajności dokumentu oraz związane z tym informacje o odbiorcy dokumentu,
- informacje pozwalające zidentyfikować zawartość dokumentu — streszczenie i/lub słowa kluczowe.

Zarządzanie dokumentacją techniczną wymaga też standaryzacji sposobu dostępu do dokumentów. Niezbędne jest stworzenie swego rodzaju biblioteki dokumentów technicznych. W niewielkiej firmie jeden z pracowników może otrzymać jako dodatkowe zadanie prowadzenie takiej biblioteki. W dużej firmie mogą zajmować się tym osoby zatrudnione specjalnie w tym celu. Biblioteka taka powinna zapewniać możliwość dotarcia do każdego dokumentu, który znajduje się w bibliotece lub jest wypożyczony przez któregoś z pracowników. Dostęp do poszczególnych dokumentów powinny uzyskiwać wyłącznie osoby do tego uprawnione.

13.6. Zarządzanie wersjami

Przekazane do eksploatacji oprogramowanie musi podlegać ciągłym zmianom (patrz rozdział 11, omawiający konserwację oprogramowania). Część z tych zmian wymaga jedynie drobnych modyfikacji. Realizacja innych może wymagać przeprowadzenia odrębnego przedsięwzięcia programistycznego. Każda modyfikacja powoduje powstanie nowej, mniej lub bardziej różniącej się od poprzedniej, wersji systemu. Z punktu widzenia firmy programistycznej najlepiej jest, jeżeli przekazanie do eksploatacji nowej wersji systemu wiąże się z usunięciem z eksploatacji poprzedniej wersji. W praktyce jednak nie wszyscy użytkownicy są zainteresowani nabyciem nowej wersji. Firma programistyczna może ich do tego zachęcać przekazując nową wersję nieodpłatnie. Ma to jednak sens ekonomiczny jedynie wtedy, gdy wprowadzone zmiany są stosunkowo niewielkie i nie pochłonęły zbyt dużych nakładów. Podejście to jest stosowane szczególnie w przypadku, gdy firma zastępuje wersję, w której wykryto błędy, wersją poprawioną.

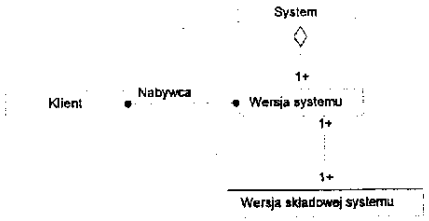
Inną przyczyną powstawania wielu wersji oprogramowania są różnorodne potrzeby poszczególnych użytkowników. Może tu chodzić o inne wymagania sprzętowe i dotyczące oprogramowania systemowego. To samo oprogramowanie może być przeznaczone do pracy w różnorodnych środowiskach sprzętowo-programistycznych. Poszczególni nabywcy mogą się też różnić wymaganiami dotyczącymi zakresu dostępnych funkcji. Poszczególne wersje mogą więc zawierać jedynie część modułów kompletnego systemu.

Zadaniem kierownictwa firmy programistycznej jest zorganizowanie odpowiedniego systemu zarządzania wersjami. Narzędzia wspomagające zarządzanie wersjami wchodzą w skład niektórych narzędzi CASE. Należy jednak przyznać, że możliwości takich nie posiada większość stosunkowo niedrogich pakietów tego typu. W takich sytuacjach niezbędne staje się stworzenie własnych lub zakup dodatkowych narzędzi wspomagających zarządzanie wersjami. Podstawą takich narzędzi jest baza danych przechowująca

informacje o eksploatowanych wersjach systemu. Powinna ona zawierać takie informacje, jak:

- informacje o wszystkich oddanych do eksploatacji wersjach, między innymi daty oddania do eksploatacji,
- informacje o klientach, którzy nabyli dana wersję,
- wymagania sprzętowe i programowe poszczególnych wersji,
- informacje o składowych (np. klasach, encjach, modułach) wchodzących w skład danej wersji,
- informacje o pozostających do realizacji żądaniach zmian,
- informacje o błędach wykrytych w poszczególnych wersjach.

Rysunek 13.6. zawiera ogólny schemat tego typu bazy danych w notacji OMT.



Rysunek 13.6. Schemat bazy danych o wersjach systemu

13.7. Miary produktywności

Ocena produktywności poszczególnych pracowników oraz zespołów programistycznych jest istotne ze względu na:

- konieczność odpowiedniego motywowania najbardziej wydajnych osób,
- możliwość wykorzystania zebranych danych podczas szacowania kosztu i czasu niezbędnego do wykonania przyszłych zadań.

Drugi z powyższych celów ma ścisły związek z możliwością osiągnięcia przez firmę wyższych poziomów dojrzałości procesu (porównaj sekcję 13.4 omawiającą poziomy rozwoju firmy programistycznej).

Tradycyjne miary produktywności stosowane w firmach programistycznych to:

- liczba linii uruchomionego i przetestowanego kodu źródłowego (bez komentarzy i linii pustych) napisanych w pewnym okresie czasu (np. w ciągu miesiąca),

- liczba instrukcji kodu wynikowego wyprodukowanych w pewnym okresie czasu,
- liczba stron dokumentacji napisanych w pewnym okresie czasu,
- liczba przykładów testowych opracowanym w pewnym okresie czasu.

Należy podkreślić, że stosowanie nowoczesnych narzędzi inżynierii oprogramowania może znacznie utrudniać lub nawet uniemożliwiać posługiwanie się powyższymi miarami. Zastosowanie języka wysokiego poziomu w miejsce języka niższego poziomu, np. wprowadzenie języka czwartej generacji, może powodować pozorny spadek liczby instrukcji kodu źródłowego wytwarzanych w tym samym czasie, choć rzeczywista wydajność pracy będzie większa. Gotowe biblioteki, generatory kodu i interfejsu użytkownika pozwalają z kolei na wyprodukowanie dużej liczby instrukcji kodu wynikowego w stosunkowo krótkim czasie. W efekcie wydajność pracowników pracujących nad fragmentami systemu, w których możliwe jest posługiwanie się tego typu narzędziami jest pozornie znacznie wyższa niż osób pracujących nad innymi składowymi. Bardzo trudna jest też ocena wydajności analityków i projektantów posługujących się głównie narzędziami CASE. Pewne miary, którymi można posługiwać się w takich sytuacjach to: liczba opracowanych klas (encji lub innych składowych), liczba opracowanych pól i metod, wielkość dokumentacji technicznej. Wiele narzędzi CASE w automatyczny sposób zachowuje informacje o dacie oraz autorze ostatniej modyfikacji danego elementu projektu. Ułatwia to stosowanie wymienionych miar produktywności.

Oprócz rodzaju stosowanych narzędzi na produktywność inżynierów oprogramowania wpływają także:

- Złożoność tworzonego systemu (porównaj dyskusję w sekcji 3.3, omawiającej model COCOMO szacowania nakładów na oprogramowanie).
- Jakość pracy w poprzednich fazach przedsięwzięcia. Źle określone wymagania utrudniają np. pracę w fazie analizy i projektowania. Zły projekt utrudnia pracę w fazie implementacji.

Powyższe czynniki są niezależne od osób, których produktywność jest oceniana. Powinno to zostać uwzględnione prze kierownictwo przedsięwzięcia, szczególnie wtedy, gdy celem oceny produktywności jest odpowiednie motywowanie pracowników.

13.8. Harmonogramowanie i monitorowanie przedsięwzięć programistycznych

Harmonogramowanie przedsięwzięcia polega na:

- ustaleniu kalendarza prac,

- podziale przedsięwzięcia na poszczególne zadania,
- określeniu parametrów zadań,
- określeniu zasobów niezbędnych do realizacji poszczególnych zadań,
- ustaleniu dostępności zasobów,
- określeniu kolejności i czasów wykonania poszczególnych zadań.

Ustalenie kalendarza prac wymaga określenia:

- daty rozpoczęcia przedsięwzięcia,
- dni roboczych i wolnych w przewidywanym okresie realizacji przedsięwzięcia,
- czasu pracy w poszczególnych dniach.

Kolejnym krokiem jest podział przedsięwzięcia na elementarne zadania. Dopuszcza się budowę hierarchii zadań, w której zadania wyższego poziomu składają się z pewnej ilości zadań poziomu niższego. Na przykład ogólne zadanie projektowanie może obejmować szereg bardziej szczegółowych zadań: projektowanie składowej zarządzania danymi, projektowanie składowej kontaktu z użytkownikiem... Hierarchia taka ułatwia edycję oraz prezentację dużej liczby zadań, jednak faktycznemu harmonogramowaniu podlegają zadania elementarne. Przedsięwzięcie powinno zostać podzielone na stosunkowo małe zadania, których parametry jest dość łatwo określić. W przypadku przedsięwzięć programistycznych są to z reguły zadania, których realizacja wymaga kilku dni. Stosowanie harmonogramowania jest więc możliwe wtedy, gdy można już dość dokładnie określić elementarne czynności niezbędne do zrealizowania przedsięwzięcia. Harmonogramowaniu nie musi podlegać całe przedsięwzięcie. W przypadku przedsięwzięć programistycznych z reguły na wstępnych etapach przedsięwzięcia nie można precyzyjnie określić przyszłych zadań z wystarczającą dokładnością, można jedynie wyróżnić pewne bardzo ogólne zadania (porównaj rozdział 3 omawiający fazę strategiczną).

Po ustaleniu zadań należy określić ich parametry czasowe oraz ograniczenia kolejności. Parametry czasowe zadań to:

- czas wykonania,
- najwcześniejszy możliwy termin rozpoczęcia, wynikający np. z tego, że wykonanie danego zadania wymaga pewnych danych, które zostaną dostarczone już po terminie rozpoczęcia przedsięwzięcia,
- pożądany czas zakończenia realizacji.

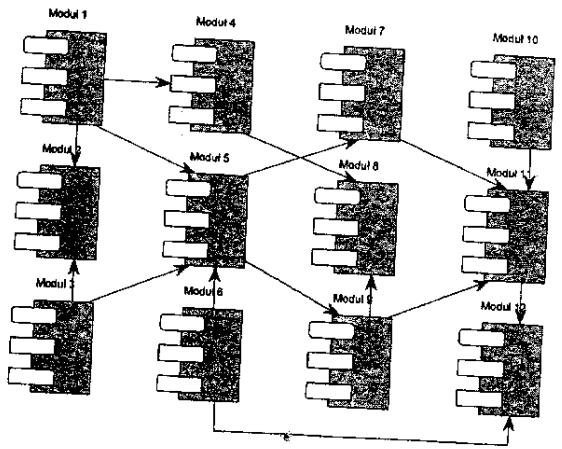
Najczęściej spotykane ograniczenia kolejności dotyczą sytuacji, w których pewne zadanie może się rozpocząć dopiero po zakończeniu realizacji jednego lub kilku innych zadań, które nazywane są jego poprzednikami. Są to ograniczenia typu koniec-początek. Inne możliwe typy ograniczeń to:

- koniec-koniec, jeżeli dane zadanie może skończyć się dopiero po zakończeniu jednego lub kilku innych zadań,

- początek-początek, jeżeli dane zadanie może rozpocząć się dopiero po rozpoczęciu realizacji jednego lub kilku innych zadań,
- początek-koniec, jeżeli dane zadanie może skończyć się dopiero po rozpoczęciu realizacji jednego lub kilku innych zadań.

Dodatkowo z każdym ograniczeniem kolejności może zostać związany czas opóźnienia lub wyprzedzenia. W przypadku ograniczenia typu koniec-początek, czas opóźnienia oznacza, że zadanie może rozpocząć się dopiero po upływie określonego czasu od zakończenia realizacji wszystkich poprzedników. Czas wyprzedzenia oznacza w tym wypadku, że zadanie może rozpocząć się na pewien czas przed zakończeniem realizacji wszystkich poprzedników.

Jako przykład zostanie rozważony problem harmonogramowania procesu implementacji dwunastu modułów przedstawionych na rysunku 13.7.



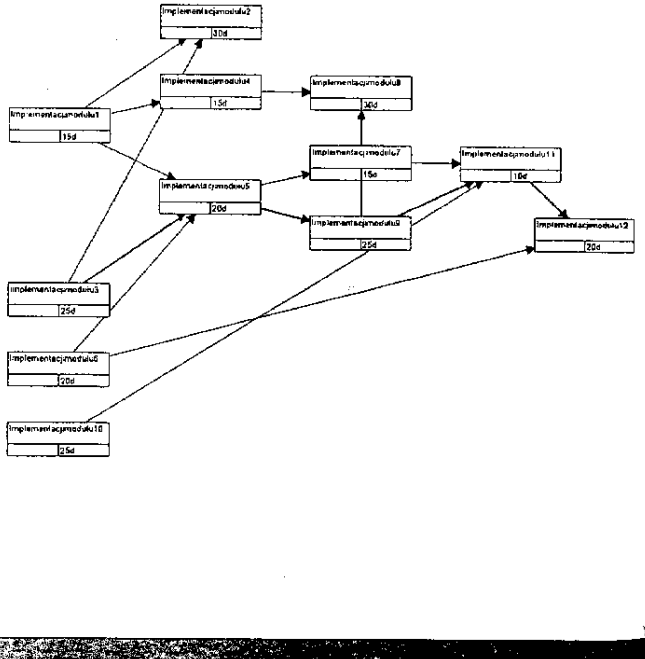
Rysunek 13.7. Przykładowy diagram zależności modułów

Implementacja i przetestowanie każdego modułu traktowana jest jako odrębne zadanie. Dla zadań tych nie określono możliwych terminów rozpoczęcia ani pożądanych czasów zakończenia. W tabeli 13.1 przedstawione są przewidywane czasy realizacji poszczególnych zadań.

Przyjęte zostało założenie, że implementacja danego modułu może rozpocząć się dopiero po zaimplementowaniu wszystkich wykorzystywanych modułów. Wynikające z tego ograniczenia kolejności przedstawione są na rysunku 13.8 na diagramie typu PERT. Ułożenie harmonogramu spełniającego ograniczenia kolejności oraz minimalizującego czas zakończenia przedsięwzięcia jest zadaniem stosunkowo prostym.

Tabela 13.1.
Zadania i ich czasy realizacji

Nazwa zadania	*Czas realizacji [dni]*
Implementacja modułu 1	15
Implementacja modułu 2	30
Implementacja modułu 3	25
Implementacja modułu 4	15
Implementacja modułu 5	20
Implementacja modułu 6	20
Implementacja modułu 7	15
Implementacja modułu 8	30
Implementacja modułu 9	25
Implementacja modułu 10	25
Implementacja modułu 11	10
Implementacja modułu 12	20



Rysunek 13.8.
Ograniczenia kolejności przykładowego problemu — diagram typu PERT

Konstrukcję takiego harmonogramu opisuje następujący algorytm:

powtarzaj

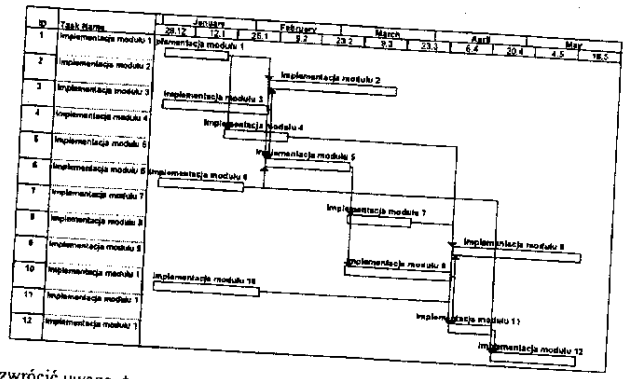
wybierz dowolne zadanie, które nie ma poprzedników, lub którego wszystkie poprzedniki mają już ustalone terminy rozpoczęcia

biorąc pod uwagę ograniczenia kolejności ustal najwcześniejszy możliwy termin rozpoczęcia wybranego zadania

dopóki nie ustalono terminów rozpoczęcia wszystkich zadań

Rysunek 13.9 przedstawia harmonogram realizacji przykładowego przedsięwzięcia w postaci wykresu Gantta. Wykres taki pozwala łatwo zobrazować terminy rozpoczęcia realizacji poszczególnych zadań oraz czas ich trwania. Na wykresie tym przedstawione są także poszczególne ograniczenia kolejności.

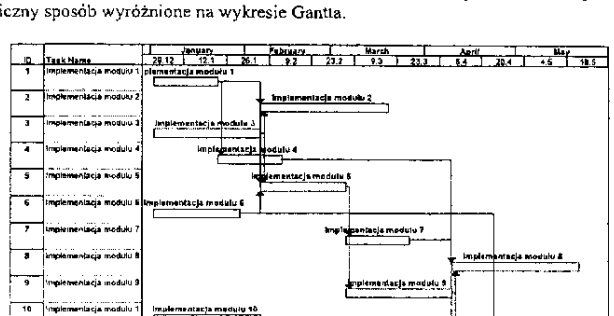
Rysunek 13.9.
Harmonogram realizacji przedsięwzięcia w formie wykresu Gantta



Warto zwrócić uwagę, że pewne zadania mogą zostać opóźnione bez wpływu na termin zakończenia całego przedsięwzięcia. Są to zadania, które nie mają następników i kończą się przed terminem zakończenia przedsięwzięcia (na przykład Implementacja modułu 2) lub zadania, których wszystkie następniki zaczynają być realizowane później niż wynika to z terminu zakończenia (i rozpoczęcia) danego zadania. Przykładowo zadanie Implementacja modułu 8 jest następnikiem zadania Implementacja modułu 7. Drugie z zadań rozpoczyna się jednak z opóźnieniem w stosunku do terminu zakończenia pierwszego z nich. Termin rozpoczęcia zadania Implementacja modułu 8 wynika z konieczności zakończenia zadnia Implementacja modułu 9. Zadanie Implementacja modułu 7 może więc zostać opóźnione w zakresie, w którym jego realizacja będzie się kończyła nie później niż realizacja zadania Implementacja modułu 9.

Zadania, które nie mogą zostać opóźnione nazywane są zadaniami krytycznymi. Ich ciąg definiujący termin zakończenia przedsięwzięcia nazywany jest ścieżką krytyczną. Zadania te powinny być przedmiotem szczególnej uwagi kierownictwa przedsięwzięcia.

W omawianym przykładzie ścieżkę krytyczną tworzą następujące zadania: Implementacja modułu 3, Implementacja modułu 5, Implementacja modułu 9, Implementacja modułu 11, Implementacja modułu 8 i Implementacja modułu 12. Na rysunku 13.10 są one w graficzny sposób wyróżnione na wykresie Gantta.



Rysunek 13.10. Harmonogram realizacji przedsięwzięcia w formie wykresu Gantta z wyróżnionymi zadaniami krytycznymi

Harmonogram przedsięwzięcia może być także przedstawiony w formie tabelarycznej. Przykład zawiera tabela 13.2. Dla każdego zadania podano najwcześniejszy oraz najpóźniejszy termin rozpoczęcia i zakończenia. Różnica najwcześniejszego i najpóźniejszego czasu rozpoczęcia nazywana jest luzem zadania. Wielkość ta może być również interpretowana jako dopuszczalne przedłużenie czasu realizacji zadania nie wpływające na termin zakończenia przedsięwzięcia. Oczywiście dla zadań krytycznych luz wynosi dokładnie 0.

Z przedstawionego na rysunkach 13.9 i 13.10 oraz tabeli 13.2 harmonogramu wynika, że w pewnych okresach będzie realizowanych kilka zadań jednocześnie. Od 02.01.97 do 29.01.97 jednocześnie realizowane będą cztery zadania. Liczba dostępnych pracowników może być jednak niewystarczająca aby zapewnić jednoczesną realizację tylu zadań. W ogólności układając harmonogram należy brać także pod uwagę dostępność zasobów niezbędnych do realizacji poszczególnych zadań. W przypadku przedsięwzięć programistycznych zasobami są przede wszystkim ludzie: analitycy, projektanci, programiści, wykonawcy testów. Zasobami mogą być także sprzęt i oprogramowanie, np. specjalistyczny pakiet CASE z ograniczoną liczbą jednoczesnych użytkowników.

W rozważanym przykładzie brane są pod uwagę zasoby wymienione w tabeli 13.3. Wymagania zasobowe poszczególnych zadań przedstawione są w tabeli 13.4.

Ułożenie harmonogramu, który brałby pod uwagę ograniczenia zasobowe oraz minimalizowałby czas zakończenia przedsięwzięcia nie jest zadaniem prostym. W narzędziach wspomagających harmonogramowanie przedsięwzięć stosuje się więc różnorodne metody przybliżone o różnej efektywności.

Tabela 13.2.
Tabelaryczny zapis harmonogramu

Zadanie	*Najwcześniejszy moment rozpoczęcia*	*Najwcześniejszy moment zakończenia*	*Najpóźniejszy moment rozpoczęcia*	*Najpóźniejszy moment zakończenia*	*Luz fdnif*
Implementacja modułu 1	02.01.97	22.01.97	16.01.97	05.02.97	10
Implementacja modułu 2	06.02.97	19.03.97	10.04.97	21.05.97	45
Implementacja modułu 3	02.01.97	05.02.97	02.01.97	05.02.97	0
Implementacja modułu 4	23.01.97	12.02.97	20.03.97	09.04.97	40
Implementacja modułu 5	06.02.97	05.03.97	06.02.97	05.03.97	0
Implementacja modułu 6	02.01.97	29.01.97	09.01.97	05.02.97	5
Implementacja modułu 7	06.03.97	26.03.97	20.03.97	09.04.97	10
Implementacja modułu 8	10.04.97	21.05.97	10.04.97	21.05.97	0
Implementacja modułu 9	06.03.97	09.04.97	06.03.97	09.04.97	0
Implementacja modułu 10	02.01.97	05.02.97	06.03.97	09.04.97	45
Implementacja modułu 11	10.04.97	23.04.97	10.04.97	23.04.97	0
Implementacja modułu 12	24.04.97	21.05.97	24.04.97	21.05.97	0

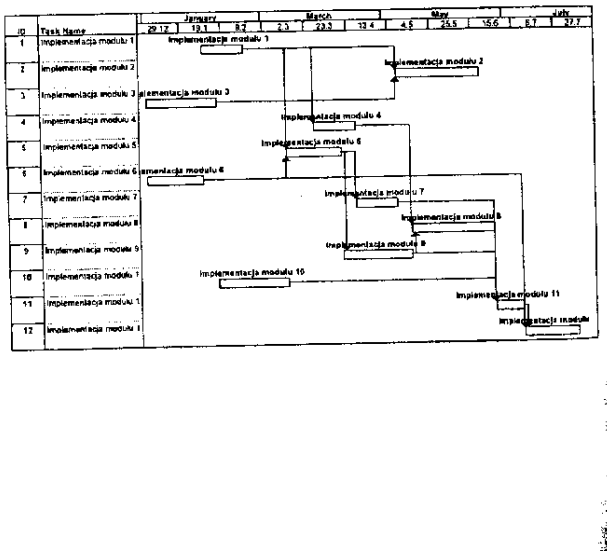
Tabela 13.3.
Zasoby przykładowego przedsięwzięcia

Rodzaj pracowników	*Liczba osób*
Projektanci	2
Programiści	4
Wykonawcy testów	2

Tabela 13.4.
Wymagania zasobowe poszczególnych zadań

Zadanie	Projektanci	Programiści	Wykonawcy testów
Implementacja modułu 1	1	2	1
Implementacja modułu 2	1	2	1
Implementacja modułu 3	1	2	1
Implementacja modułu 4	1	2	1
Implementacja modułu 5	1	2	2
Implementacja modułu 6	1	2	1
Implementacja modułu 7	1	2	1
Implementacja modułu 8	1	2	1
Implementacja modułu 9	1	2	1
Implementacja modułu 10	1	2	1
Implementacja modułu 11	1	2	2
Implementacja modułu 12	1	2	1

Rysunek 13.11 oraz tabela 13.5 zawierają harmonogram spełniający ograniczenia zasobowe przykładowego problemu.



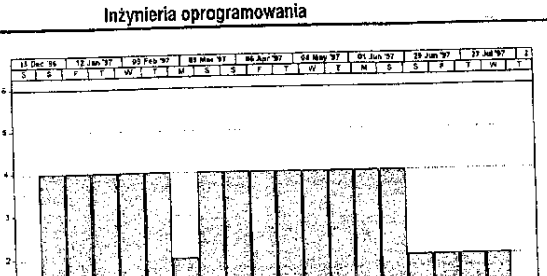
Rysunek 13.11.
Harmonogram spełniający ograniczenia zasobowe

Tabela 13.5.
Tabelaryczny zapis harmonogramu spełniającego ograniczenia zasobowe

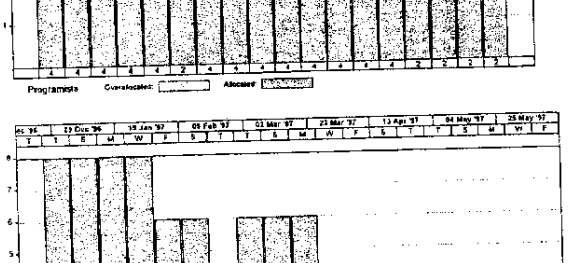
Zadanie	Najwcześniejszy moment rozpoczęcia	Najwcześniejszy moment zakończenia	Najpóźniejszy moment rozpoczęcia	Najpóźniejszy moment zakończenia	Luz [dni]
Implementacja modułu 1	30.01.97	19.02.97	03.04.97	23.04.97	45
Implementacja modułu 2	08.05.97	18.06.97	26.06.97	06.08.97	35
Implementacja modułu 3	02.01.97	05.02.97	20.03.97	23.04.97	55
Implementacja modułu 4	27.03.97	16.04.97	05.06.97	25.06.97	50
Implementacja modułu 5	13.03.97	09.04.97	24.04.97	21.05.97	30
Implementacja modułu 6	02.01.97	29.01.97	27.03.97	23.04.97	60
Implementacja modułu 7	17.04.97	07.05.97	05.06.97	25.06.97	35
Implementacja modułu 8	15.05.97	25.06.97	26.06.97	06.08.97	30
Implementacja modułu 9	10.04.97	14.05.97	22.05.97	25.06.97	30
Implementacja modułu 10	06.02.97	12.03.97	22.05.97	25.06.97	75
Implementacja modułu 11	26.06.97	09.07.97	26.06.97	09.07.97	0
Implementacja modułu 12	10.07.97	06.08.97	10.07.97	06.08.97	0

Wykorzystanie poszczególnych zasobów wynikające z powyższego harmonogramu może być zobrazowane graficznie. Przykład wykresu wykorzystania zasobów jest przedstawiony na rysunku 13.12.

Wykres wykorzystania zasobu pozwala także określić maksymalną ilość danego zasobu, którą można by przeznaczyć do realizacji danego przedsięwzięcia. Rysunek 13.13 przedstawia liczbę programistów, którą należy zatrudnić, aby zrealizować rozważane przedsięwzięcie w możliwie najkrótszym czasie. Wykres ten został więc uzyskany dla harmonogramu przedstawionego na rysunkach 13.9 oraz 13.10.



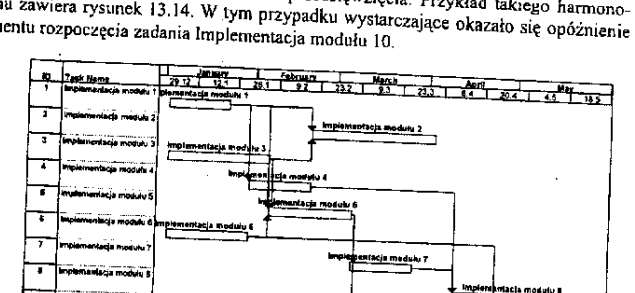
Rysunek 13.12. Wykres wykorzystania zasobu



Rysunek 13.13. Niezbędna liczba programistów w trakcie realizacji przedsięwzięcia w możliwie najkrótszym czasie

Z wykresu tego wynika, że zatrudnienie 8 programistów zapewnia wykonanie przedsięwzięcia w najkrótszym czasie. Warto jednak zauważyć, że taka liczba programistów jest potrzebna tylko w stosunkowo krótkim okresie. Przez większą część realizacji przedsięwzięcia dwóch programistów pozostanie bez zajęcia. W podobny sposób można ustalić maksymalną liczbę projektantów (4) oraz wykonawców testów (4). Wyniki te potwierdzają założenie modelu COCOMO (porównaj sekcję 3.3 omawiającą algorytmiczne modele szacowania kosztów oprogramowania), że dla każdego przedsięwzięcia istnieje optymalna liczba pracowników, którzy powinni je realizować.

Momenty rozpoczęcia zadań, które nie są zadaniami krytycznymi, mogą być przesuwane w ramach wyznaczonego luzu. Daje to szansę na zmniejszenie zapotrzebowania na poszczególne zasoby bez wydłużenia czasu realizacji przedsięwzięcia. Harmonogram taki można znaleźć ręcznie modyfikując czasy rozpoczęcia poszczególnych zadań. Prostszym wyjściem, jest próba zmniejszania ilości dostępnych zasobów i automatycznego poszukiwania harmonogramu spełniającego te zapotrzebowania. Okazuje się, że w przypadku rozważanego przedsięwzięcia 3 projektantów, 6 programistów i 4 wykonawców testów wystarcza do wykonania przedsięwzięcia. Przykład takiego harmonogramu zawiera rysunek 13.14. W tym przypadku wystarczające okazało się opóźnienie momentu rozpoczęcia zadania Implementacja modułu 10.



Rysunek 13.14. Harmonogram wymagający 3 projektantów, 6 programistów i 4 wykonawców testów

Monitorowanie przedsięwzięcia polega na śledzeniu przebiegu jego realizacji oraz reagowaniu na powstałe problemy. Na monitorowanie składają się następujące czynności:

- obserwacja rzeczywistych terminów rozpoczęcia i zakończenia zadań oraz porównywanie ich z zaplanowanymi,
- identyfikacja przyczyn niezgodności z planem,
- modyfikacja harmonogramu, jeżeli różnice pomiędzy poprzednim harmonogramem a rzeczywistym przebiegiem przedsięwzięcia są zbyt duże,
- obserwacja wykorzystania zasobów oraz rozstrzyganie konfliktów zasobowych,
- przesuwanie zasobów pomiędzy zadaniami oraz przedsięwzięciami.

Podczas monitorowania szczególną uwagę należy zwracać na zadania krytyczne oraz zadania ze stosunkowo niewielkim luzem.

13.9. Ekonomiczne aspekty działalności firmy programistycznej

W sekcji 13.2 omawiającej zakres inżynierii oprogramowania stwierdzono, że celem tej dziedziny techniki jest produkcja dobrego — zgodnego z wymaganiami użytkownika, niezawodnego, efektywnego, łatwego w konserwacji i ergonomicznego — oprogramo-

wania. Z punktu widzenia firmy programistycznej najważniejsze jest jednak osiągnięcie sukcesu ekonomicznego, czyli wysokich zysków i dużego udziału w rynku.

Jest oczywiste, że jakość produktu jest tylko jednym z czynników wpływającym na wielkość sprzedaży. Inne czynniki to między innymi:

- reklama i promocja produktu,
- renoma producenta,
- rodzaj i zakres gwarancji oraz innych usług dla klientów,
- przyzwyczajenia klientów,
- sposób wyceny rozmaitych wersji produktów.

Czynniki te często pozwalają zredukować wpływ niższej jakości produktów danej firmy. Stosowanie technik inżynierii oprogramowani, których celem jest poprawa jakości produktu wiąże się ze sporymi nakładami finansowymi. Kierownictwo firmy programistycznej musi oczywiście rozważyć, czy lepszym rozwiązaniem nie będzie przeznaczenie części nakładów na działalność marketingową firmy.

Na wielkość zysków firmy wpływają nie tylko wpływy ze sprzedaży lecz także koszty produkcji. Jak już wspomniano stosowanie technik inżynierii oprogramowania wiąże się z pewnymi kosztami. Są to na przykład:

- koszty reorganizacji firmy,
- koszty szkoleń,
- koszty zakupu narzędzi CASE,
- nakłady na dokładne testowanie oprogramowania.

Koszty te mogą być bardzo wysokie, szczególnie w firmie, która dopiero planuje zmianę sposobu pracy i przejście na wyższy poziom zaawansowania technicznego. Nakłady te należy traktować jako poważną inwestycję. Zwrotu tych nakładów można spodziewać się dopiero po pewnym czasie ponieważ:

- pracownicy będą potrzebować pewnego czasu na osiągnięcie zaawansowania w stosowaniu nowych metod i narzędzi,
- największej redukcji kosztów można się spodziewać w fazie konserwacji oprogramowania, a nie w czasie produkcji oprogramowania,
- większych wpływów ze sprzedaży lepszego jakościowo produktu można się spodziewać dopiero po zakończeniu produkcji oprogramowania.

Podjęcie decyzji o poważnej inwestycji jest istotnym problemem w każdej firmie, nie tylko programistycznej. Oczywiście zwrot nakładów nigdy nie może być traktowany jako pewny.

Literatura

Baber R.L. (1989). *O oprogramowaniu inaczej*. WNT, Warszawa, 1989.

Barker R. (1990). *CASE*METHOD. Entity relationship modelling*. Addison-Wesley, New York.

Barker R. (1990). *CASE*METHOD. Task and deliverables*. Addison-Wesley, New York.

Barker R. i Longman C. (1992). *CASE*METHOD. Function and process modelling*. Addison-Wesley, New York.

Bell T.E., Bixler D.C., Dyer M.E. (1977). An extendable approach to computer aided software requirements engineering. *IEEE Trans. Software Engineering*, **3**, 1, 49 – 60.

Bersoff E.H., Elements of software configuration management. *IEEE Trans. Software Engineering*, **10**, 1, 79 – 87, 1984.

Błażewicz J. (1988). Złożoność obliczeniowa algorytmów kombinatorycznych, WNT, Warszawa.

Booch G. (1983). *Software Engineering with Ada*. The Benjamin/Cummings Publishing Company, California.

Booch G. (1994). *Object Oriented Analysis and Design With Applications*. The Benjamin/Cummings Publishing Company, Redwood City, CA.

Brooks F. (1995). *The Mythical Man Month*. Addison-Wesley, New York, drugie wydanie.

Buhr R.J.A. (1986). *Systems Design with Ada*. Prentice-Hall, Englewood Cliffs.

Clegg D. i Barker R. (1994). *CASE*METHOD. Fast-Track*. Addison-Wesley, New York.

Coad P., Yourdon E. (1994). *Analiza obiektowa*. Read Me, Warszawa.

Coad P., Yourdon E. (1994). *Projektowanie obiektowe*. Read Me, Warszawa.

Constantine L.L., Yourdon E. (1979). *Structured Design*. Englwood Cliffs NJ, Prentice-Hall.

Cox B. (1986). *Object-Oriented Programming — An Evolutionary Approach*. Addison-Wesley, New York.

DeMarco T. (1979). *Structured Analysis and System Specification*. Prentice-Hall, Englewood Cliffs.

DeMarco T. (1982). *Controlling Software Project*. Yourdon Press, New York.

DeMarco T., Lister T. (1987). *Peopleware*. Dorset House, New York.

Dijkstra E.W. (1976). *A Discipline of Programming*. Prentince-Hall, Englewood Cliffs.

Diller A. (1990). *Z: An Introduction to Formal Methods*. John Wiley & Sons.

Dreyfus H.L., Dreyfus S.E. (1986). *Mind over machine. The power of human intuition and expertise in the era of the computer*. Basil Blackwell Ltd., Oxford.

Edwards S., King D., Winblad, A. (1990). *Object-oriented Software*. Addison-Wesley, New York.

Fagan M.E. (1986). Advances in software inspections, *IEEE Transactions in Software Engineering*, **1**, 182 – 211.

Flavin M. (1981). *Fundamental Concepts in Information Modeling*. Yourdon Press Computing Series, Englewood Cliffs, Prentice-Hall.

Frost S. (1995). *The SELECT Perspective: Developing Enterprise Systems Using Object Technology*. SELECT Software Tools, Santa Ana.

Fuglewicz P., Stąpor K., Trojan A. (1995). *CASE dla ludzi*, Lupus, Warszawa.

Gane C., Sarson T. (1986). *Structured Systems Analysis: Tools & Techniques*. McDonnell Douglas Corporation.

Genuchten van M. (1991). Why is software late? An empirical study of reasons for delay in software development. *IEEE Transactions on software engineering*, 17, 6, 582 – 590.

Ghezzi C., Jazayeri M., Mandrioli D. (1991). *Fundamentals of Software Engineering*, Prentice-Hall, Englewood Cliffs.

Humphrey W., *Managing the Software Process*, Addison-Wesley, 1989.

Hapke M., Jaszkiewicz A., Słowinski R. (1994). Fuzzy Project Scheduling System for Software Development. *Fuzzy Sets and Systems*, 67, 101 – 117.

Harres J.H. (1992). *SSADM Version 4. The advanced practitioner's guide*. Wiley, Chichester.

Hickman K., Longman C. (1995). *CASE*METHOD. Business interviewing*. Addison-Wesley, New York.

Iscoe N. (1988). *Domain Models for Program Specifications and Generation*. University of Texas, Austin.

Jacobson I. (1993). *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, New York.

Jaszkiewicz A., Matysiak M., Hapke M. (1995 – 1996). *Inżynieria oprogramowania. Materiały szkoleniowe*. Instytut Informatyki Politechniki Poznańskiej, Poznań.

Jones T.C. (1986). *Programming productivity*. McGraw-Hill, New York.

Kaliszewski M. (1994). *Inżynieria oprogramowania obiektowego. Analiza obiektowa*. Respekt, Tomaszów Mazowiecki.

Kehoe R., Jarvis A. (1995). *ISO 9000-3. A tool for software product and process improvement*. Springer, Berlin.

Kim W., Lochovsky F. (1989). *Object-Oriented Concepts, Databases, and Applications*. ACM Press/Addison-Wesley, New York.

Kurt B., Woodfield S., Embley D., (1991) *Object-Oriented Systems Analysis and Specification: A Model Driven Approach*. Prentice-Hall, Englewood Cliffs.

Levenson N. (1995). *Safeware. System safety and computers*. Addison-Wesley, New York.

Martin J. (1990). *Information Engineering Planning & Analysis*. Prentice-Hall, Englewood Cliffs.

McCabe T.J. (1983). *Structured Testing*. IEEE Computer Society Press.

McGettrick A.D. (1982). *Program Verification Using Ada*. Cambridge University Press, Cambridge.

Meyer B. (1991). *Eiffel: The Language*. Prentice-Hall, Englewood Cliffs.

Meyer B. (1988). *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs.

Miller G.A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information, *Psychological reviews*, **63**, 81 – 97.

Page-Jones M. (1980). *The Practical Guide to Structured Systems Design*. Yourdon Press Computing Series, Prentice-Hall, Englewood Cliffs.

Pankwoski T. (1992). *Podstawy baz danych*. PWN, Warszawa.

Pressman R.S. (1992). *Software Engineering. A Practitioner's Approach*. McGraw-Hill, New York.

Roy B. (1985): *Méthodologie Multicritere d'Aide a la Décision*, Economica, Paris. Tłumaczenie polskie: Wielokryterialne wspomaganie decyzji, WNT, 1990.

Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorenson, W. (1991). *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs.

Sanders J., Curran E. (1994). *Software quality. A framework for success in software development and support*. Addison-Wesley, New York.

Shlaer S., Mellor S.J. (1988). *Object-Oriented Systems Analysis: Modeling the World in Data*. Prentice-Hall, Englewood Cliffs.

Shlaer S., Mellor S.J. (1992). *Object-Oriented Systems Analysis: Modeling the World in States*. Prentice-Hall, Englewood Cliffs.

Shneiderman B. (1993). *Designing the User Interface*. Addison-Wesley, New York.

Sommerville I. (1995). *Software Engineering*. Addison-Wesley, New York.

Ward P.T., Mellor S.J. (1985). *Structured Development for Real-Time Systems*. Prentice-Hall, Englewood Cliffs.

Wirfs-Brock R., Wilkerson B., Wiener, L. (1990). *Designing Object-Oriented Software*. Prentice-Hall, Englewood Cliffs.

Yourdon E., Constantine L. (1979). *Structured Design Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Englewood Cliffs.

Yourdon E. (1989). *Modern Structured Analysis*. Prentice-Hall, Englewood Cliffs.

Słownik terminów angielskich

acceptance testing — testy akceptacji

action — akcja

activity — operacja

aggregation — związek agregacji, agregacja, związek całość-część

analysis phase — faza analizy, modelowania

association — związek (klas)

association multiplicity — krotność związku

association qualifier — kwalifikator związku

attribute — pole, atrybut (klasy, encji)

black-box tests — testy funkcjonalne

business process reengineering — reinżynieria procesów biznesowych

CASE, Computer Aided/Assisted Software/System Engineering — komputerowo wspomagana inżynieria oprogramowania

class — klasa

class link/associations — związek klas

class templates — wzorce klas

coding — implementacja, kodowanie

composition of re-usable components — montaż z gotowych elementów, programowanie z półki

data access — korzystanie z danych

data dictionary — słownik danych, repozytorium

data flow — przepływ danych

data flow diagrams — diagramy przepływów danych

data flow flag — flaga przepływu danych

data flow model — model przepływów danych

design phase — faza projektowania

directed class link/associations — skierowany związek klas

document-driven development — realizacja kierowana dokumentami

embeded project — przedsięwzięcie osadzone

encapsulation — ukrywanie danych, hermetyzacja

entity — encja

entity relation — związek encji

entity relation diagrams — diagramy związków encji

error — błąd (w programie)

event — zdarzenie

event-driven — zdarzeniocentryczna (technika modelowania przejść stanów)

event-driven programming — programowanie oparte na zdarzeniach

exploratory programming — programowanie odkrywcze

external system — system zewnętrzny

failure — błędne wykonanie

fault — błąd (w programie)

fault tree — drzewo błędów

feasibility study — studium wykonalności

field — pole, atrybut (klasy)

formal transformations model — model transformacji formalnych

function points — punkty funkcyjne

functional tests — testy funkcjonalne

generalization-specialization link/association — związek generalizacji-specjalizacji

glass-box tests — testy strukturalne

identity — tożsamość

implementation — implementacja, kodowanie

incremental development — realizacja przyrostowa

incremental programming — programowanie przyrostowe

interaction diagrams — diagramy interakcji

is-a relationship — związek generalizacji-specjalizacji

library module — moduł biblioteczny

link — związek (klas)

link multiplicity — krotność związku

message — komunikat, wywołanie metody

method — metoda, usługa, operacja (klasy)

minispecification — minispecyfikacja, pseudokod

object — obiekt

object-oriented analysis — analiza obiektowa

object-oriented design — projektowanie obiektowe

object-oriented methods — metody obiektowe

off-shell programming — programowanie z półki, montaż z gotowych elementów

operation — metoda, usługa, operacja (klasy)

organic project — przedsięwzięcie organiczne

pricing to win — wycena dla wygranej

problem domain — dziedzina problemu

problem domain analysis — analiza dziedziny problemu

process — proces

programming team — zespół programistyczny

protected access — dostęp zabezpieczony

prototyping — prototypowanie

qualified association — związek kwalifikowany

quick-and-dirty programing — szybkie programowanie

RAD, Rapid Application Development — szybkie wytwarzanie aplikacji

rapid prototyping — szybkie prototypowanie

repository — słownik danych, repozytorium

reverse engineering — inżynieria odwrotna

robustness — odporność, żywotność

robustness testing — testy odporności

semi-detached project — przedsięwzięcie półoderwane

service — metoda, usługa, operacja (klasy)

software engineering environments — środowiska inżynierii oprogramowania

software re-engineering — reinżynieria oprogramowania

software reuse — ponowne wykorzystanie oprogramowania

spiral model — model spiralny

state — stan

state-driven — stanocentryczna (technika modelowania przejść stanów)

state transition diagrams — diagramy przejść (transformacji) stanów

strategy phase — faza strategiczna

stress testing — testy pod obciążeniem

structure charts/diagrams — diagramy strukturalne

structural tests — testy strukturalne

structured analysis — analiza strukturalna

structured design — projektowanie strukturalne

structured methods — metody strukturalne

system responsibilities — zakresem odpowiedzialności systemu

transition — przejście, transformacja (ze stanu do satnu)

user interface management systems — systemy zarządzania interfejsem użytkownika

use case — przypadek (sposób) użycia (wykorzystania)

validation — atestowanie

waterfall model — model kaskadowy, model wodospadu, model liniowy

white-box tests — testy strukturalne

whole-part relationship — związek całość-część, związek agregacji

BEZPŁATNY! KATALOG

Wydawnictwo **Helion**
zaprasza do lektury katalogu swoich książek

Zamawiam bezpłatny egzemplarz najnowszego katalogu książek Wydawnictwa Helion

Zamawiający:

Dokładny adres:

Kod pocztowy:

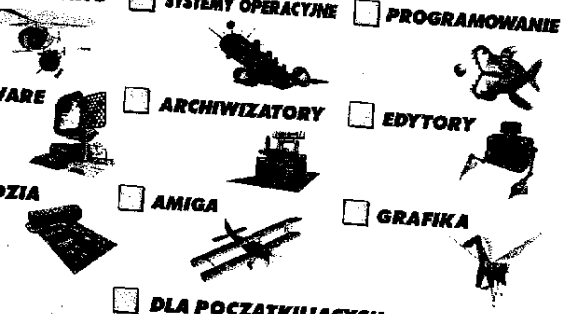
Nr telefonu:

Data i podpis

ODBIJ NA KSERO LUB WYTNIJ

INTERESUJE MNIE:

☐ CAD, RENDERING ☐ SYSTEMY OPERACYJNE ☐ PROGRAMOWANIE



☐ HARDWARE ☐ ARCHIWIZATORY ☐ EDYTORY

☐ NARZĘDZIA ☐ AMIGA ☐ GRAFIKA

☐ SIECI ☐ DLA POCZĄTKUJĄCYCH

☐ INNE

Najłatwiej zamówić przez telefon: zadzwoń i zamów!

☎ (32) 38-81-54, (32) 132-04-31

Wydawnictwo **Helion** ul. Pszczyńska 89, 44-100 Gliwice
☎ (32) 38-81-54, fax (32) 38-81-01 ✉ 44-100 Gliwice, skr. poczt. 462
e-mail: helion@silesia.ternet.pl