Partner:

WSPÓŁCZESNE TECHNOLOGIE INFORMATYCZNE

METODYKI ZWINNE

WYTWARZANIA OPROGRAMOWANIA

Projekt Absolwent na miarę czasu współfinansowany przez Unię Europejską
w ramach Europejskiego Funduszu Społecznego
Nr umowy UDA-POKL.04.01.01-00-421/10-01

Politechnika Lubelska
Wydział Elektrotechniki i Informatyki
ul. Nadbystrzycka 38A
20-618 Lublin

WSPÓŁCZESNE TECHNOLOGIE INFORMATYCZNE

METODYKI ZWINNE

WYTWARZANIA OPROGRAMOWANIA

Marek Miłosz

Magdalena Borys

Małgorzata Plechawska-Wójcik

Politechnika Lubelska
Lublin 2011

Recenzenci:
dr inż. Maria Skublewska-Paszkowska
dr inż. Piotr Muryjas

Skład komputerowy: Marek Miłosz

Publikacja finansowana z projektu „Absolwent na miarę czasu"

Projekt „Absolwent na miarę czasu" współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego. Nr umowy UDA-POKL.04.01.01-00-421/10-01

Ta publikacja odzwierciedla jedynie stanowiska jej autorów, a Komisja Europejska nie ponosi odpowiedzialności za informacje w niej zawarte

Publikacja dystrybuowana bezpłatnie

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-62596-61-4

Wydawca: Politechnika Lubelska
ul. Nadbystrzycka 38D, 20-618 Lublin
Realizacja: Biblioteka Politechniki Lubelskiej
Ośrodek ds. Wydawnictw i Biblioteki Cyfrowej
ul. Nadbystrzycka 36A, 20-618 Lublin
tel. (81) 538-46-59, email: wydawca@pollub.pl
www.biblioteka.pollub.pl
Druk: ESUS Agencja Reklamowo-Wydawnicza Tomasz Przybylak
www.esus.pl

Elektroniczna wersja książki dostępna w Bibliotece Cyfrowej PL www.bc.pollub.pl
Nakad: 100 egz.

Spis treści

Wstęp

Zwinność w procesach wytwarzania oprogramowania związana jest z chęcią zmiany podejścia do procesu wytwórczego, odrzuceniem pewnych elementów formalizmu metodyk tradycyjnych oraz wykorzystaniu najlepszych praktyk pracy zespołowej, zwiększających jego efektywność i jakość rezultatu. Dynamizuje ona proces wytwarzania oprogramowania, wykorzystuje efekt synergii wynikający ze ścisłej współpracy z klientem w trakcie całego projektu, umożliwia dynamiczne sterowanie przyrostowym wytwarzaniem oprogramowania przy pomocy zmieniających się wymagań klienta.

Podejście zwinne zaowocowało takimi metodykami realizacji przedsięwzięć informatycznych jak programowanie ekstremalne, Scrum czy FDD. Zmieniło także podejście do procesów szacowania, planowania i kontroli realizacji przedsięwzięcia.

Jakość oprogramowania jest jednym z fundamentów podejścia zwinnego. Specjalne techniki tworzenia kodu, wielopoziomowe testowanie oraz standaryzacja rozwiązań to tylko niektóre techniki jej osiągnięcia. Zaangażowanie zdopingowanego zespołu, poczucie własności i równocześnie współdzielenie kodu jest dodatkowym czynnikiem wzrostu jego jakości, w tym i podatności na modyfikacje.

Standaryzacja rozwiązań, pozwalająca nie odkrywać ponownie „koła” i stosować sprawdzone elementy oprogramowania, zaowocowała zdefiniowaniem (częściej zwykłą klasyfikacja i opisem) wzorcowych konstrukcji, zwanych wzorcami

projektowymi. Ich stosowanie rewolucjonizuje podejście do konstruowania architektury oprogramowania i upraszcza komunikacje w zespole programistów.

Wielokrotne poprawianie kodu programu jest jedną ze specyficznych cech podejścia zwinnego. W wyniku takich operacji w kodzie mogą pojawić się „brzydkie zapachy", czyli błędy lub tylko miejsca, w których te błędy mogą się potencjalnie pojawić. By je usunąć powstała metoda refaktoryzacji, która częściowo automatyzowana pozwala poprawiać jakość kodu równolegle z jego rozwojem, realizowanym iteracyjnie.

Książka prezentuje metody zwinne wytwarzania oprogramowania. Nie jest to jednak pełnia wiedzy w tym temacie. Z jednej strony bowiem, ograniczona objętość nie pozwoliła zawrzeć wielu elementów zwinności (głównie przykładów zastosowań), a z drugiej – jest to dość szybko zmienna dziedzina informatyki. Czytelnik posiądzie wiedzę o istocie i podstawach technik zwinnych, co pozwoli mu na j ej dalsze samodzielne zgłębianie, teoretyczne i praktyczne.

Autorzy wyrażają swoje podziękowania recenzentom, których uwagi niewątpliwie przyczyniły się do poprawy jakość tej książki

Autorzy

1

Wybrane metodyki zwinne w projektach informatycznych

Cel

Rozdział poświęcony jest problemom zarządzania projektami informatycznymi ze szczególnym uwzględnieniem metodyk zwinnych. Zawiera on analizę przyczyn powstania metodyk zwinnych i manifest zwinności, czyli zbiór wartości oraz praktyk zwinnych. Przedstawione zostały szczegółowo najbardziej znane metodyki: XP, Scrum i FDD. W końcu rozdziału zaprezentowane zostały podstawowe problemy, związane z wdrożeniem metodyk zwinnych w firmach programistycznych, zagrożenia tego procesu oraz najważniejsze zalety. Zaprezentowano także rezultaty badań nad stanem zastosowań praktyk i metodyk zwinnych.

Plan

1. Projekty informatyczne i tradycyjne metodyki zarządzania.
2. Manifest zwinności i metodyki zwinne wytwarzania oprogramowania.
3. Programowanie ekstremalne i jego praktyki.
4. Scrum w wytwarzaniu oprogramowania.
5. Metodyka FDD.
6. Wdrażanie praktyk zwinnych.

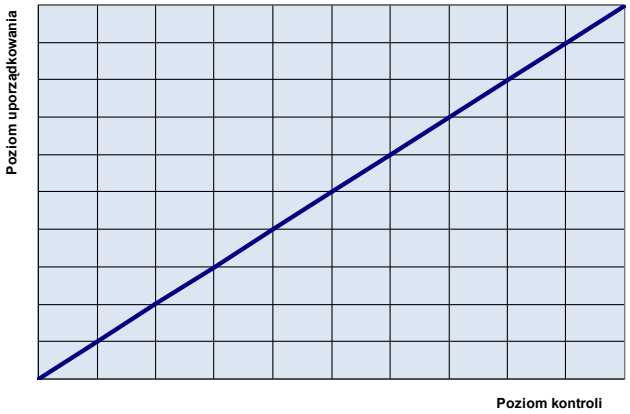
1.1. PROJEKTY INFORMATYCZNE I TRADYCYJNE METODYKI ZARZĄDZANIA

W latach 70-80 XX wieku w obszarze wytwarzania oprogramowania uwidocznił się kryzys. Przejawiał się on małą liczbą projektów informatycznych ukończonych z sukcesem. Wzrastały także gwałtownie koszty utrzymania już wdrożonego oprogramowania. Koszty te wynikały ze złej jakości oprogramowania (kodu), co znacznie zwiększało koszty jego modyfikacji i rozbudowy. Innym problemem wynikającym ze złej jakości kodu były błędy wykonania, które powodowały wymierne straty finansowe użytkowników. Badanie tego zjawiska rozpoczęto już w 1987 roku, ale bardziej kompleksowe analizy są prowadzone od 1994 roku przez Stadnish Group (Czarnacka-Chrobot, 2004). Pozwoliły one ocenić ilościowo zjawisko - okazało się, że tylko 16% projektów zakończyło się sukcesem. Przy czym sukces był rozumiany w bardzo uproszczony sposób: jako utrzymanie trzech najważniejszych parametrów projektu na zaplanowanym wcześniej poziomie. Te trzy wyróżniki sukcesu to: czas realizacji projektu, koszt i zgodność dostarczonego produktu z pierwotną specyfikacją.

Niska jakość projektów objawiała się niedotrzymywaniem terminów ich ukończenia, przekroczeniem zaplanowanego budżetu oraz niezgodnością oprogramowania z postawionymi wymaganiami. Udział projektów, które przekroczyły czas i budżet realizacji, wahał się wokół 70% i powoli maleje (Czarnacka-Chrobot, 2004). W tym samym czasie (tj. w latach 1994-2002) jednak wzrastał procent niezrealizowanych wymagań w projektach produkcji oprogramowania - z około 40% do ponad 50% (Czarnacka-Chrobot, 2004). Zjawisko kryzysu w obszarze wytwarzania oprogramowania zostało nazwane „chaosem".

Reakcją na chaos w realizacji projektów informatycznych było wprowadzenie porządku w obszarze wytwarzania oprogramowania. Powstała nowa dziedzina informatyki zwana inżynierią oprogramowania, sformalizowano i częściowo zestandaryzowano artefakty i procesy projektowo-wytwórcze (Miłosz, 2006). Oprogramowanie zaczęło być postrzegane jako zwykły produkt inżynierski, taki jak dom, most czy samochód. Równocześnie zaczęto porządkować procesy wytwórcze, wprowadzając systemowe metody zarządzania projektami do procesów wytwarzania

oprogramowania. Wierzono, że uporządkowanie procesów (a więc ograniczenie zidentyfikowanego chaosu w projektach informatycznych) można osiągnąć poprzez wprowadzenie szczegółowego planowania zadań i rygorystycznej kontroli ich wykonania. Z rezultatów badań Stadnish Group wyprowadzona została bowiem uproszczona zależność (rys. 1.1): „im większy poziom kontroli (formalizmu, sterowania, nadzoru, sprawdzania, śledzenia, reagowania na bieżąco) tym większy poziom uporządkowania (ograniczenia „chaosu"), a w konsekwencji większe prawdopodobieństwo osiągnięcia sukcesu" (Miłosz, 2006) projektu informatycznego.



Rys. 1.1. Wyidealizowana zależność pomiędzy poziomem kontroli a uporządkowaniem procesów w projektach informatycznych
Źródło: (Miłosz, 2006)

W celu zwiększenia stopnia przewidywalności i kontroli realizacji procesów wytwarzania oprogramowania zaczęto stosować tradycyjne (klasyczne) metodyki zarządzania projektami (przedsięwzięciami), znane z innych obszarów działalności ludzkiej. Metodyki tradycyjne charakteryzują się dużą restrykcyjnością, wysokim poziomem formalizmu i dbania o kontrolę procesów. Przewidują bowiem ścisłą kontrolę większości procesów, zarówno planistycznych, jak i realizacyjnych czy rozliczeniowych. „Metodyki ukierunkowane są zatem na kontrolowanie (a w zasadzie restrykcyjne zarządzanie) zakresu, czasu, zasobów, ludzi, budżetu, zmian, komunikacji, jakości i ryzyka" (Miłosz, 2006).

Tradycyjne metodyki zarządzania projektami informatycznymi powstawały w różnych miejscach i ośrodkach, w związku z czym mają różne korzenie. Można je pogrupować następująco:

- Metodyki **ogólne**, uniwersalne, tj. takie które znajdują zastosowanie w wielu obszarach aktywności, nie tylko dla projektów informatycznych. Do nich zalicza się przede wszystkim amerykańską metodykę PMI (ang. *Project Management Institute*), brytyjską - APM (ang. *Association for Project Management*), japońską - P2M (ang. *Project & Program Management for Enterprise Innovation*), a także podejście Kepner Tregoe'a.
- Metodyki **firmowe**, tj. takie które powstały w laboratoriach poszczególnych firm informatycznych i zostały upublicznione a nawet przyjęte jako standardy. Są to metodyki firmy IBM pod nazwą PMM (ang. *Project Management Methodology*), brytyjska metodyka PRINCE2 (ang. *PRojects IN Controlled Environments*) pierwotnie opracowana dla Central Computer and Telecommunication Agency (CCTA) oraz RUP (ang. *Rational Unified Process*) opracowana przez firmę Rational Software (obecnie jest to dział IBM). Są one ukierunkowane na zarządzanie projektami informatycznymi, realizowanymi dla konkretnego klienta w pełnym cyklu wytwarzania (przy wykorzystaniu różnych modeli cyklów życia: od kaskadowego do iteracyjnego) i wdrożenia, ze szczególnym uwzględnieniem dużych projektów.
- Metodyki **specjalne**, które są ukierunkowane na specyficzne typy projektów informatycznych (zarówno produkcji jak i wdrożenia oprogramowania). Są to m.in. MSF (ang. *Microsoft Solution Framework*) firmy Microsoft dla realizacji projektów wytwarzania oprogramowania oraz ASAP (ang. *Accelerated SAP*) firmy SAP dla projektów wdrożenia systemów klasy ERP (ang. *Enterprise Resource Planning).*

Niektóre metodyki firmowe z czasem stały się metodykami ogólnymi. Taka metamorfoza miała miejsce z metodyką PRINCE2, która z czasem stała się powszechnie obowiązująca w Wielkiej Brytanii i całej Europie (Koszlajda, 2010).

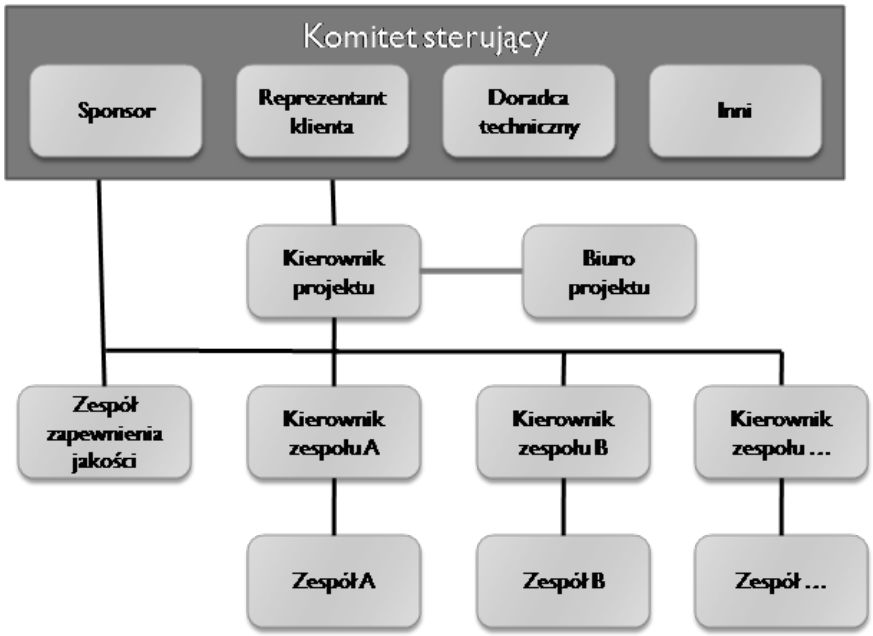
Odrębnym typem metodyk zarządzania projektami informatycznymi są metodyki wdrażania systemów informatycznych (Flasiński, 2008). Pomimo wspólnych cech (są to typowe metodyki tradycyjne) metodyki wykorzystywane w procesach wdrożeniowych są bardzo specyficzne. Są one bardzo silnie związane z wdrażanym

oprogramowaniem oraz realizowanymi procesami dla konkretnego klienta końcowego. Można je zatem zaliczyć do metodyk firmowych. Są bowiem opracowywane przez dostawców oprogramowania i przez nich wykorzystywane. Do metodyk wdrożeniowych zalicza się (Milosz, 2003) następujące (w nawiasach podane zostały nazwy wdrażanych systemów):

- ASAP firmy SAP (system R3 czy mySAP),
- AIM firmy IFS (IFS Applications),
- Implex - Intentia-Vimex (system Movex),
- JBA Advantage - JBA (system JBA System 21),
- Q-Managment firmy QAD (system MFG/Pro),
- MEGA - Quantum (system MEGA),
- Epicor i Kewill - Qumak (system Platinum/MMRP),
- Signature firmy Scala (system Scala Global Series),
- Target - TCH Systems (system Baan IV),
- Extract II - Exact Software (system Exact),
- REP - Normax (system JDEdwards),
- Fourth Shift firmy Hogart (system Fourth Shift),
- Microsoft Business Solutions Partner Methodology firmy Microsoft (system MBS Axapta).

W czasie realizacji projektów wdrożeniowych stosowane są, ale dość rzadko, także metodyki ogólne.

Tradycyjne metodyki zarządzania projektami formalizują strukturę zadaniową, zasobową oraz organizacyjną projektu. Typowa struktura zadaniowa projektu składa się z następujących, realizowanych zwykle kaskadowo etapów (Miłosz, 2003): identyfikacja, planowanie, realizacja i ukończenie projektu. Metodyki opisują i formalizują procesy oraz procedury ich realizacji na wszystkich etapach projektu (Koszlajda, 2010). Podobnie, definiują struktury organizacyjne, tj. sposoby zorganizowania pracy ludziom-członkom zespołów realizujących projekty. Są to w większości wielopoziomowe, zhierarchizowane struktury organizacyjne, składające się z Komitetu sterującego, Kierownika projektu, Zespołów wykonawczych, uwzględniające takie specjalne komórki jak Zespół zapewnienia jakości czy Biuro projektu (rys. 1.2).



Rys. 1.2. Typowa struktura organizacyjna w tradycyjnych metodykach zarządzania projektami
Źródło: opracowanie własne

Metodyki tradycyjne zarządzania projektami wprowadziły znaczną formalizację całego procesu jego realizacji. Głównym celem było ograniczenie chaosu. Formalizacja ta postrzegana jest bardzo często negatywnie poprzez pryzmat nadmiernej i zbędnej pracy organizacyjno-dokumentacyjnej, zmniejszenia elastyczności decyzyjnej (jako naturalny rezultat zbyt precyzyjnego zaplanowania procesów) oraz ograniczenia zdolności do wprowadzania zmian w trakcie trwania projektu. Ma to swoje uzasadnienie w dużych (sięgających niemal 30-40%) narzutach na pracochłonność w trakcie realizacji projektów metodykami tradycyjnymi. Nadmierna biurokratyzacja projektów stała się obiektem powszechnej krytyki. Przykładowo, Yourdon zwraca uwagę, że „nadmiar formalności zmniejsza wydajność i powoduje niepotrzebne opóźnienia. Im większa jest firma, tym biurokracja jest bardziej rozbudowana" (Yourdon, 2000).

Jako reakcja na zbyt restrykcyjne procesy, „niepotrzebne" tracenie sił i środków, ograniczenie elastyczności w działaniach oraz zbyt długotrwałe oczekiwanie na

rezultat projektu przy zmieniających się wymaganiach powstały w informatyce metodyki zwinne (ang. *Agile*), zwane niekiedy lekkimi. Metodyki te uwzględniają cechy swoiste projektom wytwarzania oprogramowania i są trudne do wykorzystania w innych obszarach ludzkiej działalności. Możliwe jest jednak stosowanie niektórych zwinnych metodyk i praktyk w projektach konsultingowych czy organizacyjnych (Schwaber, 2005). Ich istotę wyraża Manifest zwinności (ang. *Manifesto for Agile Software Development*) - http://agilemanifesto.org.

1.2. MANIFEST ZWINNOŚCI I ZWINNE METODYKI WYTWARZANIA OPROGRAMOWANIA

Manifest zwinności (pełna nazwa: Manifest zwinnego wytwarzania oprogramowania) to deklaracja filozofii realizacji projektów programistycznych. Jest to deklaracja wartości, które wg. jego autorów są najważniejsze przy opracowywaniu oprogramowania. Autorami deklaracji jest 17. twórców oprogramowania, stosujących inne niźli kaskadowe podejście do procesu jego rozwoju. Poza wartościami manifest zawiera 12 zasad, które tworzą fundament zwinnego podejścia.

Manifest zwinności, powstały w 2001 roku w USA, zawiera następującą główną treść (Shore, Warden, 2008):

„Odkrywamy lepsze sposoby rozwoju oprogramowania, wytwarzając je i pomagając w tym innym. Dzięki temu nauczyliśmy się cenić:

- Jednostki i interakcje bardziej niż procesy i narzędzia.
- Działające oprogramowanie bardziej niż kompletną dokumentację.
- Współpracę z klientem bardziej niż negocjowanie kontraktu.
- Reagowanie na zmiany bardziej niż postępowanie zgodne z planem.

Oznacza to, że choć pozycje po prawej stronie są ważne, bardziej cenimy zagadnienia przedstawione po lewej stronie."

Poza wartościami deklarowanymi w Manifeście zwinności, zarówno preambuła jak i uwaga końcowa są bardzo ważne do zrozumienia jego istoty. Nie neguje on bowiem całkowicie ważności procesów, narzędzi, dokumentacji czy planów, jak to się niestety

często akcentuje w różnych publikacjach. Manifest wskazuje na to, co jest w podejściu zwinnym bardziej cenione, a ni e co nie jest stosowane. Jednym słowem podejście zwinne nie odrzuca całkowicie planów i kontraktów, procesów i dokumentacji, dyscypliny planistycznej i wykonawczej Są one ważne, ale dla twórców oprogramowania nie najważniejsze.

Manifest zwinności został uzupełniony o zasady stojące u jego podstaw, a mianowicie (Shore, Warden, 2008):

„Stosujemy się do następujących zasad:

- Naszym priorytetem jest zapewnienie satysfakcji klientów poprzez szybkie i ciągłe udostępnienie wartościowego oprogramowania.
- Nie obawiamy się zmian wymagań nawet na późnych etapach rozwoju. Zwinne procesy pozwalają wykorzystać zmiany do zapewnienia klientom przewagi konkurencyjnej.
- Często udostępniamy działające oprogramowanie (w odstępach od kilku tygodni do kilku miesięcy), przy czym preferujemy jak najkrótsze terminy.
- Odbiorcy i programiści muszą regularnie współpracować ze sobą w czasie trwania projektu.
- Budujemy projekty wokół zmotywowanych jednostek. Należy zapewnić im środowisko i zaspokajać potrzeby oraz ufać, że wykonają powierzone zadania.
- Najbardziej wydajna i efektywna metoda przekazywania informacji zespołowi programistycznemu i w obrębie niego to rozmowy twarzą w twarz.
- Działające oprogramowanie to główny wyznacznik postępów.
- Zwinne procesy promują zrównoważony rozwój. Sponsorzy, programiści i użytkownicy powinni móc zachować stałe tempo pracy.
- Ciągłe poświęcanie uwagi technicznej doskonałości i dobremu projektowi zwiększa zwinność.
- Niezwykle istotna jest prostota, czyli sztuka maksymalizowania liczby zadań, których nie trzeba wykonywać.
- Najlepsze architektury, wymagania i projekty są efektem pracy samodzielnie organizujących się zespołów.
- W stałych odstępach czasu zespół określa, jak może zwiększyć wydajność, a następnie w odpowiedni sposób usprawnia i dostosowuje swe działania.”

Zasady zwinności zalecają zatem stosowanie technik programowania iteracyjnego (przyrostowego), realizowanego w krótkich kompletnych cyklach, które dostarczają:

- funkcjonujące,
- poprawne (duże znaczenie ma zatem testowanie produktu),
- proste w swojej konstrukcji (standaryzowane, bez stosowania skomplikowanych konstrukcji i trików)
- cenne dla klienta (tj. dostarczającej właściwej funkcjonalności, bez zbędnych fajerwerków, odpowiednio wydajne)

oprogramowanie.

Taki model wytwarzania oprogramowania zapewnia zwinność w reakcji na zmiany wymagań, ale nie jest niczym nowych w inżynierii oprogramowania.

Istotnym elementem podejścia zwinnego, silnie odróżniającego go od tradycyjnych metodyk realizacji projektów, jest sposób traktowania zespołu i organizacji jego pracy. Pojawiające się zasady stałej współpracy wszystkich osób zaangażowanych w projekt oraz zwiększenia swobody pracy zespołu programistów całkowicie zmieniają strukturę organizacyjną, techniki pracy i zarządzania projektem (Cockburn, 2008). Zasady współpracy osób w czasie projektu zawierają bowiem następujące postulaty:

- samodzielnej organizacji zespołu
- współodpowiedzialności całego zespołu za rezultat pracy,
- silnego zmotywowania (samo-motywowania) członków,
- zaspokojenia potrzeb członków zespołu,
- obdarzenia zespołu dużym zaufaniem.

Dochodzi do tego zasada współpracy bezpośredniej (z pominięciem komunikacji formalnej, dokumentów) wymuszająca z jednej strony pracę synchroniczną w jednym pomieszczeniu (lub też stosowanie częstych interakcji przy pomocy wideokonferencji), a z drugiej promująca niewielkie (5-9 osobowe) zespoły programistów. W przypadku dużych projektów programistycznych zasada ta wymusza podział ich na mniejsze fragmenty realizowane przez oddzielne, ale stale komunikujące się zespoły, pod nadzorem zespołu koordynującego (Krebs, 2009).

Byciu zwinnym pomagają takie metody i techniki jak: ustawiczna integracja kodu, automatyczne testowanie, programowanie parami, rozwój sterowany testami (ang.

Test Driven Development, TDD), wzorce projektowe (ang. *Design Patterns*), refaktoryzacja i kontrola wersji kodu, historyjki użytkownika, gra planistyczna czy retrospekcje.

Do najważniejszych metodyk zwinnych zarządzania projektami informatycznymi zalicza się następujące:

- Programowanie ekstremalne (ang. *eXtreme Programing*, w skrócie *XP*),
- Scrum,
- FDD (ang. *Future-Driven Development*),
- DSDM (ang. *Dynamic Systems Development Method*),
- Crystal Clear,
- AUP (ang. *Agile Unified Process*),
- XPrince (ang. *Extreme Programming in Controlled Environments*).

Metodyki XP, Scrum i FDD, jako najczęściej stosowane, będą szczegółowo przedstawione w następnych rozdziałach.

Metodyka **DSDM** ukierunkowana jest na dotrzymanie ustalonego terminu realizacji projektu, przy dopuszczeniu możliwości zmiany wymagań i zakresu w trakcie jego realizacji. Zakłada ona również aktywny udział klienta w trakcie realizacji projektu oraz pełnię władz decyzyjnych zespołu realizującego (ang. *DSDM Team*), a także aktywne wykorzystanie metod prototypowania, iteracyjności i przyrostowości procesu dostarczania produktu. Głównymi iteracjami są: modelowanie funkcji, programowanie oraz jego wdrożenie, rozumiane jako przekazanie produktu do użytkowania. Metodyka przewiduje stosowanie zasady „good enough" (a właściwie zasady Pareto: 80/20), wyrażającej się w fundamentalnej dla metodyki regule, że dostarczenie produktu na czas jest ważniejsze od jego kompletności (a więc zakresu i jakości). W konsekwencji, szczególnej uwadze w trakcie projektu poświęcane jest najważniejszym funkcjom systemu (około 20% ze wszystkich), które to muszą być dostarczone na czas z poziomem błędów umożliwiającym ich wykorzystanie. Pozostałe funkcjonalności powinny, ale nie muszą być dostarczane w komplecie. Niedokończone lub zawierające zbyt dużo błędów funkcjonalności nie są uwidoczniane w interfejsie dostarczanego produktu.

Crystal Clear jest metodyką, której podstawowym założeniem jest: „opracowanie oprogramowania jest zespołową grą innowacyjności i komunikacji" (Cockburn, 2004).

Gra ta wyraża się częstymi dostawami produktu do użytkownika (dostawy bazują na cyklu miesięcznym, tj. 30. dniowym), wykorzystaniu dotychczasowych doświadczeń (podejście adaptacyjne, wykorzystujące refleksję nad dotychczasowym przebiegiem prac) przy produkcji następnych przyrostów, permanentnej i nieskrępowanej komunikacji w zespole (przy czym niezbyt licznym) przy zapewnieniu bezpieczeństwa osobistego (zapewnienie stabilnych i dobrych warunków zatrudnienia) członkom zespołu oraz swobodnego dostępu do ekspertów. Metodyka zakłada również aktywne wykorzystanie narzędzi CASE, takich jak automatyczne testowanie. Wprowadza częste iteracje oraz zarządzanie konfiguracją powstającego oprogramowania.

Metodyka **AUP** (ang. *Agile Unified Process*) jest wzorowana na metodyce RUP (ang. *Rational Unified Process*). Zrezygnowano w niej z części artefaktów projektowych, dyscyplin i procesów RUP, a także ról członków zespołu RUP zachowując przy tym iteracyjność procesu, jego 4. etapowość oraz zapewnienie kontroli jakości dostarczanego produktu poprzez testowanie (ang. *Test Driven Development, TDD*).

XPrince (ang. *Extreme Programming in Controlled Environments*) jest metodyką opracowana przez zespół prof. J.Nawrockiego z Politechniki Poznańskiej (Meyer, 2007). Powstała ona w wyniku twórczego połączenia XP i metodyki PRINCE2. Jest ona metodyką zwinną, zawierającą jednak elementy kontroli przedsięwzięcia (w aspektach jakości, ryzyka, pracy) oraz niezbędny poziom wymaganej dokumentacji (z wykorzystaniem niektórych diagramów UML). W XPrince struktura organizacyjna projektu łączy zalety obu metodyk, zapewniając nadzór nad projektem przy minimalnych kosztach zarządzania.

Inne, mniej znane metodyki lekkie mają podobne zasady, wynikające z Manifestu zwinności, a mianowicie:

- ukierunkowanie na produkt oraz jego dynamiczną zmianę,
- wykorzystanie modeli iteracyjnych wytwarzania oprogramowania (wiele powtarzających się identycznych etapów),
- podejście adaptacyjne (wykorzystanie na bieżąco zdobytej wiedzy w trakcie realizacji projektu),
- ukierunkowanie na małe, dynamiczne i samoorganizujące się zespoły.

Twórcy wielu metodyk zwinnych sami ograniczają ich stosowalność do projektów:

- elastycznych w obszarze procesowym i wykonawczym,
- realizowanych przy pomocy niewielkiego zespołu,
- o znanych na początku, w ogólnych zarysach, wymaganiach do produktu.

W wielu przypadkach nie jest możliwe swobodne podejście do planów, terminów, zakresu czy też budżetu, nawet jak jest ono zgodne z interesami klienta. Takie ograniczenie może wynikać z wewnętrznych zasad budżetowania przedsięwzięć u klienta, uwarunkowań zewnętrznych (np. ustawa o zamówieniach publicznych), bądź też jest narzucone przez stronę współfinansującą projekt (np. projekt finansowany poprzez system grantów).

Inne słabe strony metodyk zwinnych są oczywiste. Ograniczają się one bowiem do projektów o niewielkiej skali, realizowanych bardzo dynamicznie (a więc w małym horyzoncie czasowym). Dostarczony produkt nie posiada dokumentacji, a jedynie spełnia wymagania klienta, który to niestety musi być silnie zaangażowany w proces wytwórczy.

Niewielka skala projektu, zarządzanego metodyką zwinną, wynika z założenia realizacji go pr zez niewielkie, samoorganizujące sie i niezależne (a więc nie zorganizowane w struktury hierarchiczne) zespoły w dość krótkich cyklach wykonawczych. Trudno bowiem zorganizować elastyczne i dynamicznie, samoorganizujące się bardzo duże, wieloosobowe zespoły.

Poza tym dynamika procesu oraz jego adaptacyjność silnie ogranicza horyzont planowania projektu zwinnego. W niektórych przypadkach brak długookresowego planowania jest niedopuszczalny lub nieakceptowalny przez klienta, który mimo wszystko (tj. pomimo stałej współpracy z twórcami systemu oraz bieżących konsultacji) chciałby (lub musi) wiedzieć co i na kiedy będzie zrealizowane lub dostarczone.

Brak dokumentacji oprogramowania może w znacznym stopniu podnieść koszty jego serwisowania oraz rozwoju, pomimo stosowanej w większości przypadków zasady prostoty i standaryzacji oraz technik refaktoryzacji kodu. W niektórych przypadkach dokumentacja produktu ma kolosalne znaczenie lub wręcz jest wymagana prawem (np. systemy bankowe), bądź też dobrymi praktykami (Flasiński, 2008).

Istotnym problemem, który pojawia się w wielu projektach informatycznych, jest brak możliwości spełnienia jednego z podstawowych założeń wszystkich metodyk zwinnych, a mianowicie dużego i ustawicznego zaangażowanie klienta w proces wytwórczy. Bardzo często klient jest pasywny, nie ma czasu, wiedzy i umiejętności, czy jest wręcz przeciwnikiem realizacji projektu. Ustawiczne konsultowanie i dostarczanie do testowania coraz to nowych przyrostów oprogramowania w takich warunkach nie jest możliwe. Klient nie kontroluje procesu wytwórczego, nie uczestniczy w nim i w konsekwencji projekt żyje własnym życiem, niekoniecznie spełniając wymagania klienta. Wymagania te w takim przypadku pozostają ukryte, nierozpoznane lub też błędnie zidentyfikowane.

1.3. PROGRAMOWANIE EKSTREMALNE I JEGO PRAKTYKI

Metodyka XP (pol. *Programowanie ekstremalne*) została zaproponowana w 1999 roku, a więc jeszcze przed powstaniem Manifestu zwinności, i opublikowana przez Kenta Becka (Beck, 1999). Jest to metodyka realizacji projektów programistycznych ukierunkowana na podnoszenie jakości oprogramowania i na polepszenie stopnia dostosowania go do zmieniających się potrzeb klientów. Jej celem przewodnim jest zatem orientacja na spełnienie potrzeb biznesowych klienta. XP ciągle ewoluuje i dlatego trudno jest nazwać go całkowicie wykrystalizowaną, ustabilizowaną, wyspecyfikowaną oraz zestandaryzowaną metodyką. Programowanie ekstremalne należy rozpatrywać raczej jako zbiór dobrych praktyk, które stosowane są w sposób uzależniony od konkretnego kontekstu realizacji oprogramowania. Ich wykorzystanie zawsze jest jednak ukierunkowane na spełnienie istotnych wartości metodyki.

XP bazuje na kilku wartościach fundamentalnych w procesie tworzenia oprogramowania (pierwotnie zdefiniowane zostały cztery, ale potem rozszerzono ich listę do pięciu). Istotne wartości w XP są następujące:

- **komunikacja** – ścisła wymiana informacji z użytkownikiem i wewnątrz zespołu;
- **prostota** – koncentrowanie się na najprostszych rozwiązaniach, standardowy i prosty sposób kodowania, unikanie niepotrzebnych działań oraz produktów;

- **informacyjne sprzężenie zwrotne** (ang. *Feedback*) – spływ informacji do wszystkich członków zespołu, pochodzącej od systemu (np. rezultat testowania), użytkowników (np. opinie, uwagi) oraz zespołu (np. dane o zmianach czy zagrożeniu); informacja ta jest wykorzystywana do dos konalenia procesów (np. szacowania) oraz produktów (np. interfejsu);
- **odwaga** – śmiałość w podejmowaniu decyzji projektowych oraz komunikowaniu ewentualnych problemów;
- **wzajemny szacunek** w stosunku do wszystkich uczestników projektu oraz każdego uczestnika w stosunku do samego siebie; każdy uczestnik projektu winien być odpowiedzialny, uczciwy, lojalny oraz jednocześnie wysłuchany i doceniany.

Praktyki XP (techniki, metody) są stosowane na różnych etapach cyklu życia oprogramowania, w taki sposób by spełnić w/w istotne wartości metodyki. Prace realizuje zespół w określonym cyklu.

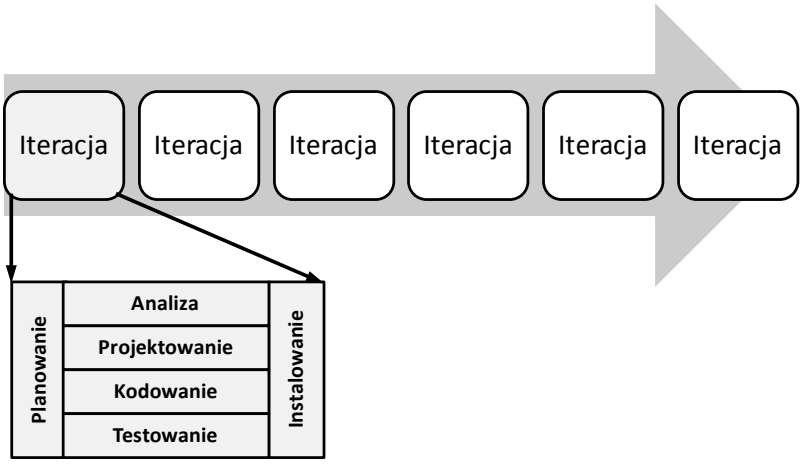
Zespół w metodyce XP generalnie składa się z dwóch grup osób, zwanych klientami i programistami. Zespoły XP nie są zbyt liczne – składają się z 4-10 programistów. Metodyka zaleca także zaangażowanie dostatecznie dużej liczby klientów. Shore J. i Warden S. (Shore, Warden, 2008) wskazują na optymalny stosunek liczby przedstawicieli klientów do programistów na poziomie 2:3.

Klienci to przedstawiciele zamawiającego. Klientem może być firma zamawiająca oprogramowanie lub dział firmy, odpowiedzialny za strategię rozwoju oprogramowania produkowanego na szeroko rozumiany rynek zbytu. Klienci są odpowiedzialni za wizję produktu, przygotowują wymagania (w tym i szczegółowe) do oprogramowania, podejmują decyzje w imieniu klienta, są odpowiedzialni za pozyskanie informacji potrzebnych programistom oraz za weryfikację poprawności produktów projektu. Klienci muszą aktywnie uczestniczyć w pracach projektowych. Grono klientów składa się z menedżera produktu, ekspertów dziedzinowych, projektantów interakcji i analityków biznesowych. Jedną z istotnych osób spośród klientów jest **menedżer (właściciel) produktu**. Ma on jedno zadanie: zarządzanie wizją produktu i propagowanie jej (Shore, Warden, 2008). Poza pracą w projekcie, menedżerowie produktu zwykle odpowiadają także za wprowadzenie produktu na rynek, a więc za reklamę, promocję, szkolenia itd. **Eksperci dziedzinowi** wnoszą do projektu wiedzę

z danego obszaru. To oni są najbardziej aktywnymi członkami zespołu XP i spędzają najwięcej czasu z zespołem. **Projektanci interakcji** definiują scenariusze działania użytkownika końcowego z systemem. Aktywnie współpracują zarówno z zespołem jak i użytkownikiem końcowym. **Analitycy biznesowi** wspomagają zespół i są łącznikiem pomiędzy pozostałymi klientami, użytkownikami końcowymi i programistami.

Programiści w zespołach XP realizują oprogramowanie – wszyscy tworzą kod. Tym niemniej, pojawiają się różne potrzeby, które realizowane są przez programistów o różnych rolach (i w konsekwencji o różnej wiedzy i umiejętnościach) Są to: projektanci, architekci, specjaliści do spraw technicznych i interfejsu, testerzy i, jeśli zachodzi taka potrzeba, graficy. Zespół programistów XP musi bowiem wykonywać bardzo różnorodne prace, takie jak analiza, projektowanie, programowanie, testowanie itd. Tak duża różnorodność i uniwersalność członków zespołu wynika z tego, że skład zespołu się nie zmienia w trakcie prac, a wszyscy kolektywnie odpowiadają za produkt. W składzie zespołu XP pojawia się także menedżer projektu i coacher (trener, osoba wspomagająca innych w rozwoju ich kompetencji). **Coachowie** wspomagają członków zespołu w wykorzystaniu ich potencjału, tworzą warunki do efektywnej pracy, są osobami doświadczonymi i szanowanymi. **Menedżer projektu** wspomaga zespół w obszarze współpracy z otoczeniem (np. macierzystą firmą). W przypadku produkcji oprogramowania na rynek menedżer projektu jest klientem wewnętrznym, czyli menedżerem produktu.

Cykl życia w XP składa się z wielu, następujących po sobie **iteracji**. W każdej iteracji występują etapy, które w znaczniej części są realizowane równolegle. Klasycznie stosowana jest iteracja o długości jednego tygodnia, ale można ją wydłużyć nawet do trzech tygodni. Po początkowym zaplanowaniu iteracji (tj. wyboru, co w danej iteracji ma być zrealizowane), zespoły wytwarzające oprogramowanie równolegle wykonują etapy analizy, projektowanie, kodowania i testowania. Iterację kończy instalacja (udostępnienie) nowej wersji oprogramowania (rys. 1.3). Instalacja ta jest realizowana zwykle na potrzeby wewnętrzne (w celu prezentacji rezultatów iteracji), a tylko od czasu do czasu na potrzeby użytkownika końcowego.

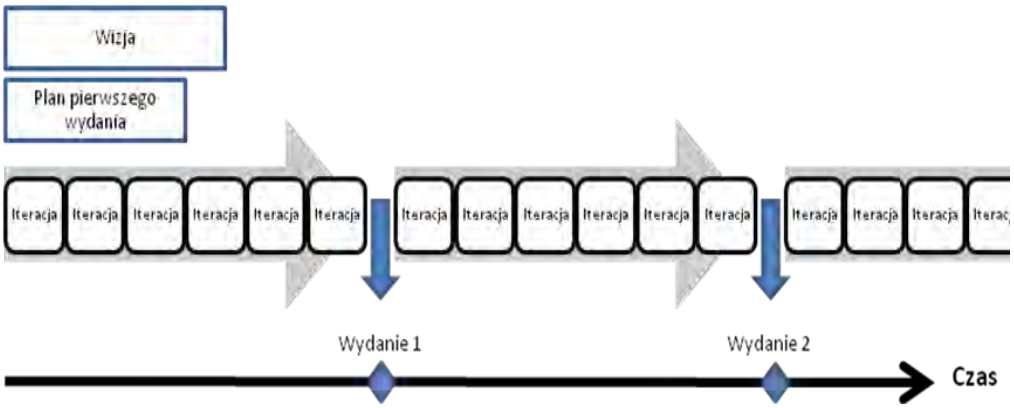


Rys. 1.3. Cykl życia jednego wydania w XP
Źródło: (Shore, Warden, 2008)

Grupa iteracji kończy się **wydaniem**. Wydanie (ang. *Release*) to oprogramowanie dostarczane, instalowane oraz wdrażane u klienta, bądź też dostarczane w wersji do sprzedaży w sieci dystrybucji. Przy czym XP zakłada pełną automatyzację przekazywania kolejnych wydań (tj. bez zbędnego angażowania zespołu w ten proces). Czas pomiędzy wydaniami może być różny i jest silnie uwarunkowany potrzebami biznesowymi klienta (zwykle jest to kwartał, ale niekoniecznie). Całość projektu rozpoczyna opracowaniem **wizji** projektu, czyli ogólnych biznesowych i funkcjonalnych założeń, które powinny być osiągnięte w wyniku realizacji projektu. Wizja może być tworzona częściowo równolegle z początkowymi iteracjami pierwszego wydania. Po ukończeniu opracowania wizji (lub równolegle z tym procesem) planowane są zwykle daty wydań oraz określany szczegółowo zakres funkcjonalny jednego lub dwóch pierwszych wydań. Cykl ten przedstawiony został na rys. 1.4.

Projekt realizowany metodyką XP posiada zatem pewne wbudowane formalizmy (np. iteracje czy wydania), ale też jest niezwykle elastyczny. Wizja jest bowiem dokumentem bardzo ogólnym, a zakres prac, które będą realizowane w ciągu najbliższego czasu ustalany jest sukcesywnie w miarę (i na podstawie doświadczeń)

realizacji poprzednich etapów. Może on być także w każdej chwili zmodyfikowany. Takie podejście wymaga jednak respektowania istotnych wartości XP przez wszystkie strony projektu.



Rys. 1.4. Cykl życia oprogramowania w XP
Źródło: opracowanie własne

W XP początkowo zdefiniowanych zostało 12 praktyk (Beck, 1999), których użycie powoduje spełnienie istotnych wartości XP. W kolejnych wydaniach książki Kenta Becka liczba ta wzrosła, a inni autorzy dodawali do XP dodatkowe praktyki, które wynikały z ich doświadczeń. Przykładowo w książce Shore i Wardena (Shore, Warden, 2008) można doliczyć się już ponad 50 praktyk, a i tak o kilku z nich autorzy tylko wspominają, że je pominęli z powodu braku doświadczeń. Podobnie sprawa się ma z podziałem praktyk na grupy. Ich taksonomia jest bowiem bardzo różna. Do najważniejszych i najbardziej znanych praktyk stosowanych w metodyce XP należą (Martin, 2008):

- Opowieści (historyjki) użytkownika.
- Gra planistyczna.
- Prostota projektu.
- Refaktoryzacja (przebudowa) kodu.
- Programowanie parami.
- Standaryzacja kodowania.

- Współwłasność kodu.
- Programowanie poprzedzane testami.
- Ciągła integracja.
- Dziesięciominutowa kompilacja.
- Wspólny język (Metafora).
- Częste wydania.
- Wspólna praca.
- Energiczna praca (40-godzinny tydzień pracy).

Niektóre z praktyk XP będą omówione poniżej, a inne (jak np. refaktoryzacja kodu) – w następnych rozdziałach, z racji ich dużej złożoności.

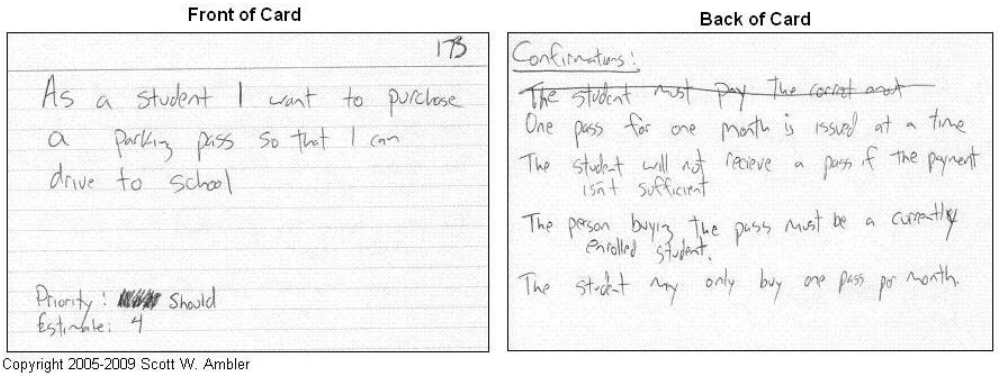
Opowieści (historyjki) użytkownika (ang. *User Stories*), zwane niekiedy kartami wymagań, służą do definiowania wymagań funkcjonalnych do oprogramowania. Są to proste jednozdaniowe opisy tego co zespół ma wytworzyć, zdefiniowane z punktu widzenia klienta, np. „Egzaminator ma mieć możliwość wprowadzenia ocen z egzaminu", „Za egzamin poprawkowy student może zapłacić przelewem lub kartą płatniczą". Nie są one kompletnymi wymaganiami, ale opisem elementów oprogramowania, budowy których klient oczekuje od zespołu programistów. Nie zawierają one szczegółów implementacyjnych, chociaż niektóre mogą być blisko wymagań niefunkcjonalnych, np. „Lista ocen ma być dostępna przy pomocy zwykłej przeglądarki internetowej". Opowieści powinny być tak tworzone by na realizację elementu oprogramowania, wynikającego z jednej opowieści zespół programistów poświęcał od jednego do dwóch dni pracy. Po uzupełnieniu o priorytet dla klienta i oszacowanie czasu realizacji służą one jako podstawa do planowania szczegółowego (w trakcie tzw. gry planistycznej). Za definiowanie wymagań, a więc za przygotowanie opowieści, są odpowiedzialni klienci. Historyjki użytkownika powinny mieć dwie ważne cechy, a mianowicie:

- Mają określoną wartość dla klienta i powinny być napisane w jego języku (tj. z użyciem słownictwa klienta). Opisują one efekt, a nie sposób realizacji. Nie są też opisem elementów bez znaczenia dla klienta chociaż ważnych dla programistów.
- Posiadają jasne, mierzalne kryteria realizacji i oceny poprawności.

Zapisując wymagania w postaci historyjek należy trzymać sie zasady INVEST (Dajda, 2008). Dobre opowieści powinny być:

- niezależne (ang. ***I**ndependent*),
- negocjowalne (ang. ***N**egotiable*),
- wartościowe dla odbiorcy (ang. ***V**aluable for Users and Customers*),
- możliwa do oszacowania (ang. ***E**stimatable*),
- małe (ang. ***S**mall*),
- testowalne (ang. ***T**estable*).

Niekiedy opowieść użytkownika jest formalizowana w postaci schematu: *Jako <aktor> chcę <czegoś> by <zysk>*, np. „Jako pracownik dziekanatu chcę wydrukować listę ocen studentów z egzaminu by ją wywiesić w gablocie" lub „Jako klient chcę dodać produkt do k oszyka by go potem kupić". Użyte, w zdaniu określającym opowieść, czasowniki (chcę, musi, powinno, może, dobrze by było itd.) służą do ustalania ich priorytetów. Każda pojedyncza opowieść zapisywana jest na pojedynczej karteczce (najlepiej samoprzylepnej) i może mieć nadany unikalny identyfikator. Zbiór tych karteczek umieszczony na dużej tablicy używany jest potem w procesie planowania wydania, czyli grze planistycznej. Karteczki mogą być również używane do zapisywania uzgodnień (np. uszczegółowienie, opis wyjątków itd.) związanych z daną opowieścią (rys. 1.5).



Rys. 1.5. Karteczka z opowieścią użytkownika

Źródło: (http://www.agilemodeling.com/artifacts/userStory.htm)

Gra planistyczna (ang. *Planning Game*) to sposób postępowania w celu zaplanowania poszczególnych wydań projektu. Jest to zespołowe działanie ukierunkowane na maksymalizację wartości dla klienta elementów, dostarczanych w planowanym okresie, przy minimalizacji kosztów. Wiedza o wartości biznesowej elementów należy do klienta, a o kosztach – do programistów. Z uwagi na ograniczony czas wydania i możliwości wytwórcze zespołu programistów przyjęcie do realizacji jednego elementu projektu oznacza zwykle rezygnację z innego. Rezultatem gry jest grupa opowieści użytkownika (a więc funkcji oprogramowania), która będzie realizowana i dostarczone w danym wydaniu. W czasie gry klienci zajmują się wyborem opowieści i określaniem ich priorytetów, a programiści szacowaniem kosztów realizacji opowieści (więcej o metrykach szacowania kosztów wykonania, tj. dniach, dniach idealnych, punktach opowieści użytkownika itd., - w następnym rozdziale). Przy czym żadna z grup nie tworzy oddzielnego planu, a współpracują, udostępniając sobie wzajemnie wiedzę o wszystkich elementach projektu. Gra przebiega według następującego scenariusza (nie jest to jedyny scenariusz; możliwe są także działania iteracyjne):

1. Tworzenie opowieści użytkownika (i/lub wybieranie ich z istniejącego zbioru jeszcze niezrealizowanych opowieści) do potencjalnej realizacji w danym wydaniu – realizują wszyscy, ale główne idee pochodzą od klienta.
2. Szacowanie kosztów każdej z opowieści – realizują programiści przy współpracy z klientami.
3. Szeregowanie opowieści, zgodnie z ich wartościami dla klienta – realizują klienci we współpracy z programistami.
4. Budowa planu, czyli wypełnienie budżetu możliwości (zwykle: czasu pracy programistów) w danym wydaniu przez opowieści zgodnie z przyjętym w pkt. 3. porządkiem – wspólna praca całego zespołu.

W trakcie gry planistycznej może dochodzić do świadomego przenoszenia opowieści użytkownika do późniejszego wydania, zmiany ich, a nawet usunięcia oraz do pojawienia się nowych. Decydujące zdanie w tym zakresie ma klient. Programiści mogą doradzać klientom w tym zakresie (zwykle uwzględniając możliwości techniczne oraz czasowe realizacji opowieści). Trwają także dyskusje na temat poprawności oszacowań i przydziału priorytetów. Wszystkie decyzje zespół podejmuje na zasadzie

konsensusu. Trzymanie się podstawowych wartości XP przez wszystkich uczestników gry powoduje, że przyjęte plany są rzetelne i, w miarę realizacji kolejnych wydań, coraz bardziej dokładne. Przyczyniają się do tego przede wszystkim informacyjne sprzężenie zwrotne, odwaga i wzajemny szacunek.

Podstawową metodą organizacji pracy zespołowej w grze planistycznej jest duża biała tablica oraz karteczki z opowieściami użytkownika. Na tablicy można swobodnie przemieszczać opowieści, grupować je itd. Tablica zawsze jest w zasięgu wzroku całego zespołu w trakcie gry, jak i potem w czasie realizacji projektu.

Gra planistyczna i jej rezultat silnie odróżniają się od tradycyjnych metod planowania działań w metodykach formalnych (wykorzystujących zwykle diagramy Gantta i metody sieciowe). Koncentruje się ona bowiem na zaplanowaniu tego co zespół ma wytworzyć w trakcie wydania, a nie na tym jakie konkretne zadania mają być zrealizowane i przez kogo. Zespół potem może sam zorganizować pracę w taki sposób by zakończyć ją w terminie. Odbywa się to na etapie planowania iteracji (rys. 1.3).

Planowanie iteracji również odbywa się przy pomocy gry planistycznej. Uczestniczą w niej tylko programiści. W jej trakcie poszczególne opowieści użytkownika są przekształcane do pojedynczych zadań, a każde zadanie jest szacowane pod kątem czasu jego realizacji i przydzielane poszczególnym programistom (lub ich parom) z uwzględnieniem ich rzeczywistego budżetu czasowego w danej iteracji. Planowanie iteracji zbliża to XP do metodyk formalnych (przynajmniej w zakresie pojedynczej iteracji).

Prostota projektu przejawia się w realizacji zadań w możliwy najprostszy sposób i ustawicznej rezygnacji z elementów uniwersalnych, ogólnych, zbyt skomplikowanych, trikowych itd. „Prosty to nie to samo co uproszczony" (Shore, Warde, 2008). Innymi słowy oznacza to, że można stosować niebanalne rozwiązania, ale je trzeba prezentować w sposób czytelny i zrozumiały z użyciem **wspólnego języka** lub ogólnie zrozumiałych **metafor**. Według Kenta Becka prostota wyraża się poprzez:

- Dostosowanie elementów do odbiorców. Nie jest ważne jak wspaniała jest realizacja, ważne jest by było zrozumiała dla tych co z nią muszą pracować (dotyczy to zarówno klientów jak i programistów).

- Komunikatywność. Każdy pomysł, idea, działanie musi być jasno przedstawione.
- Uporządkowanie. Brak porządku utrudnia zrozumienia i nadmiernie komplikuje elementy projektu.
- Minimalność. Produkt i sam projekt powinien zawierać jak najmniejszą liczbę artefaktów przy uwzględnieniu powyższych cech. Mała liczba elementów ułatwia zrozumienie, testowanie, opisanie, poznanie, wdrożenie itd.

Prostota oznacza także realizację tylko tego (i w takim zakresie) co zostało określone dzisiaj, bez realizacji elementów „na wszelki wypadek". Dość żartobliwie ideę prostoty wyraża myśl: jest to maksymalizacja liczby rzeczy, których nie trzeba robić w projekcie.

Na prostotę składają się następujące szczegółowe techniki (Shore, Warde, 2008): unikanie pisania kodu potencjalnie potrzebnego, ale nie niezbędnego teraz (ang. *You Aren't Gonna Need It, YAGNI*), brak dublowania się elementów (zakaz stosowania: ang. *Copy-Paste*), samodokumentujący się kod (a więc wykorzystywanie odpowiednich identyfikatorów i konstrukcji ograniczające konieczność użycia komentarzy w kodzie), izolowanie niezależnych komponentów (np. z wykorzystaniem wzorca projektowego Adapter – patrz rozdz. 4) oraz ograniczenie liczby udostępnianych interfejsów. Te dwa ostatnie ograniczenia mają na celu uproszczenie wprowadzania zmian do oprogramowania. Twórcy oprogramowania zawsze mają na względzie konieczność wprowadzania zmiany. W XP zmiana ta jest jednak na tyle nieokreślona i nieprzewidywalna, że nie warto poświęcać czasu na jej i tworzyć rozwiązań na przyszłość. Tym niemniej w trakcie realizacji projektu kod może być wielokrotnie poprawiany. Fakt ten związany jest z iteracyjnym cyklem tworzenia oprogramowania. By zachować prostotę kodu trzeba go stale modyfikować – przebudowywać. Proces ten określa praktyka **refaktoryzacji**, polegająca na przekształceniu kodu w celu jego uproszczenia, poprawienia struktury i przejrzystości, a także poprawienia wydajności. Każda refaktoryzacja kończy się testowaniem oprogramowania.

Programowanie parami jest najbardziej rozpoznawalną praktyką XP. Polega ono na tworzeniu kodu równolegle przez dwóch programistów, ale z wykorzystaniem jednego komputera (terminala). W parze jedna osoba w danej chwili tworzy kod a druga obserwuje, kontroluje i wymyśla rozwiązania. Osoba tworząca kod

koncentruje się na poprawnym tworzeniem bieżącego kodu bez rozpraszania się na temat planowania całości programu, w tym czasie druga koncentruje się nad zagadnieniami strategicznymi. Para tworzy kod lepszej jakości i pracuje wydajniej. Para programistów jest także mniej narażona na przerywania, wynikające z działań innych członków zespołu. Takie przerwanie „obsługuje" jeden członek pary, a drugi w tym czasie kontynuuje pracę. Para także załatwia większość problemów między sobą, generując mniej przerwań skierowanych do innych członków zespołu, co powoduje podniesienie jego łącznej wydajności. Członkowie pary dość często zamieniają się rolami. Podobnie, w zespole co jakiś czas skład par jest zmieniany. Organizacja stanowiska pracy pary programistów powinna umożliwiać swobodną pracę dwóch osób z jednym (najlepiej dużym) monitorem.

Para programistów komunikuje się poprzez kod (łatwo bowiem prześledzić logikę myślenia partnera obserwując zastosowane konstrukcje). Unikaniu trywialnych problemów komunikacyjnych (i nie rozpraszaniu się na nich) pomaga **standaryzacja kodowania** oraz stosowanie wzorców projektowych. W parach wiedza przepływa bardzo szybko pomiędzy programistami. Zmienność składu par powoduje wyrównywanie (w kierunku podnoszenie) poziomu wiedzy i umiejętności całego zespołu. Przyczynia się także w naturalny sposób do współwłasności kodu.

Współwłasność kodu oznacza, że żaden programista nie posiada wiedzy na temat dowolnego fragmentu tworzonego oprogramowania, której to nie posiadałby inny programista. Przyczynia się do tego również wspomniana już praktyka standaryzacji kodowania. W takiej sytuacji każdy programista dość łatwo może podjąć pracę nad dowolnym fragmentem kodu. Sam kod jest bowiem tworzony w świadomości, że ktoś inny będzie musiał go modyfikować za jakiś czas. Programista XP musi się pozbyć naturalnego poczucia własności kodu na rzecz odpowiedzialności zespołu za tworzony produkt. Praktyka współwłasności kodu sprzyja bezpieczeństwu biznesowemu klienta – modyfikacje oprogramowania są zawsze możliwe (niezależnie od dostępności tego czy innego programisty), a ich koszty ograniczane są do minimum.

Programowanie poprzedzane testami (ang. *Write Tests First*) jest jedną z najbardziej nielubianych przez programistów praktyk XP. Zgodnie z nią programiści powinni przygotować przypadki testowe (**testy jednostkowe**) przed rozpoczęciem

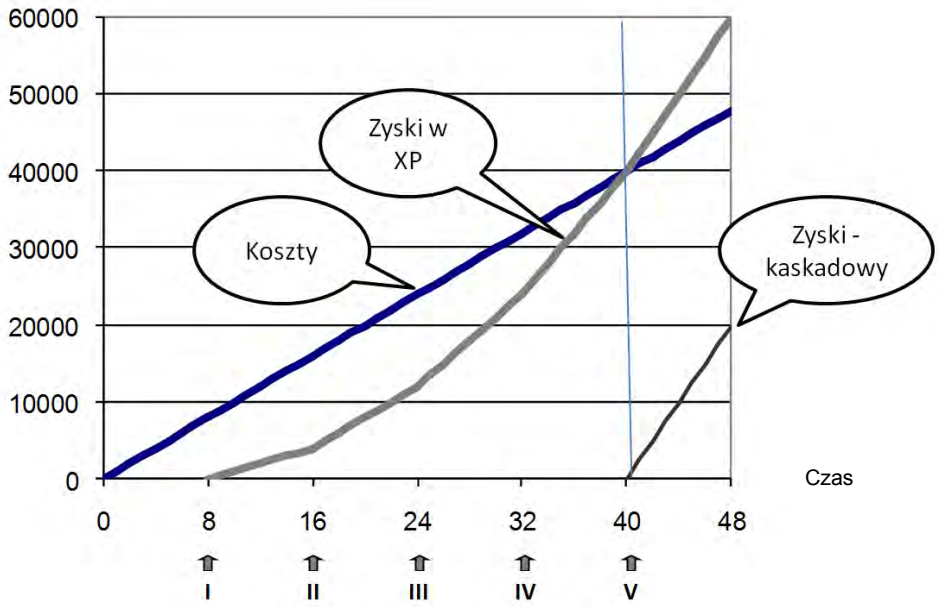
kodowania danej funkcjonalności. Zestawy testów będą potem wykorzystane do sprawdzenia poprawności opracowanego kodu. Testy powinny być jednostkowe (tj. związane tylko z realizowaną funkcjonalnością) oraz w pełni zautomatyzowane. Praktyka ta wynika z często obserwowalnego przy produkcji oprogramowania faktu: pod koniec projektu brak jest czasu i zasobów na właściwe przetestowanie systemu, a liczba błędów, które muszą być poprawione, a nie zostały wcześniej wykryte, jest bardzo duża. Wcześniejsze przygotowanie pakietu testów oraz ich użycie do sprawdzenia poprawności zmodyfikowanego kodu ma zapobiegać takim sytuacjom. Przypadki testowe powinny być aktualizowane w przypadku zmiany w opowieściach użytkownika lub odkrycia nieprzewidzianego w nich błędu. Poza testami jednostkowymi, klienci przygotowują **testy akceptacyjne**, wykonywane po zakończeniu każdego wydania i badające system pod kątem zgodności z opowieściami użytkownika. Testy akceptacyjne rzadko udaje się automatyzować. Wymaganie automatyzacji procesu testowania jednostkowego związane jest z kolejną praktyką XP, a mianowicie ciągłą integracją.

Ciągła integracja to praktyka mówiąca o konieczności codziennej kompilacji i integracji całego systemu (ang. *Daily Build*). Rezultat codziennej pracy zespołu programistów jest integrowany do systemu i poddawany automatycznemu testowaniu (w tym, w wersji idealnej, również testowaniu akceptacyjnemu). Następnego dnia programiści otrzymują rezultaty testów i w przypadku pojawienia się błędów proces ich poprawienie ma najwyższy priorytet.

Dziesięciominutowa kompilacja jest istotną praktyką, która ogranicza straty czasu programistów. Kompilacja oprogramowania jest wykonywana bowiem wielokrotnie w ciągu dnia. Kompilacja powinna się zatem odbywać automatycznie (zgodnie z przygotowanym wcześniej skryptem kompilacji) i powinna dawać jednoznaczny rezultat (tj. udana/nieudana). **Skrypty kompilacji** (a tak *de facto* integracji) powinny być zawsze aktualne. Za ich aktualizację odpowiada każdy programista i cały zespół. Kompilacja zawsze jest powiązana z realizacją testów jednostkowych. Należy zatem zadbać by testy były przeprowadzane jak najszybciej, tak by cały proces nie przekroczył 10. minut.

Częste wydania są praktyką umożliwiającą utrzymanie stałego rytmu realizacji projektu i dostarczanie klientowi oprogramowania w momentach, zgodnych z jego

wymaganiami biznesowymi. Częstotliwość wydań jest planowana na początku projektu i zwykle wynosi jedno na 1-3 miesięcy. Przekazanie wydania klientowi nie powinno angażować dodatkowych zasobów zespołu XP. Powinno być zatem bezobsługowe, w pełni automatyczne.



Rys. 1.6. Skumulowane koszty wytworzenia oprogramowania i zyski z jego wykorzystania
Źródło: opracowanie własne

Częste wydania, w porównaniu do dostarczenia oprogramowania w rezultacie projektu realizowanego cyklem kaskadowym, są efektywniejsze ekonomicznie dla klienta. Uproszczoną i bardzo poglądową analizę ekonomiczną tego problemu przedstawia rys. 1.6. Zobrazowano na nim skumulowane koszty realizacji projektu (są one proporcjonalne w czasie z uwagi na stałą liczbę członków zespołu) oraz skumulowane zyski w projektach realizowanych metodyką XP i w klasycznym cyklu kaskadowym. Oś x została wyskalowana w czasie i przedstawiono na ni ej kolejne wydania oprogramowania (I, II, III, …). Przy identycznych kosztach zyski z eksploatacji oprogramowania pojawiają się po każdym wydaniu przy stosowaniu metodyki XP (przy czym, w związku ze zwiększającą się funkcjonalnością

oprogramowania, są one coraz intensywniejsze) i po zakończeniu projektu przy podejściu kaskadowym (założono sytuację w podejściu kaskadowym, że funkcjonalności oprogramowania będą identyczne z tymi po V wydaniu w XP). Różnice są oczywiste.

Wspólna oraz **energiczna praca (**pierwotnie praktyka ta nosiła nazwę: **40-godzinny tydzień pracy)** to praktyki XP związane z organizacją pracy zespołu. Są one ukierunkowane na utrzymanie kondycji zespołu na poziomie gwarantującym jego wysoką wydajność oraz na to by cały zespół aktywnie uczestniczył w realizacji projektu, posiadał wiedzę o nim i czuł się kolektywnie odpowiedzialny za dostarczany produkt. Praktyki te wprowadzają wymaganie by zespół nie pracował długotrwale w nadgodzinach (co jest niestety dość częstą praktyką) oraz wymuszają taką realizację prac w ciągu dnia roboczego by przeciwdziałać naturalnemu zmęczeniu (np. poprzez okresowe przerwy w pracy). W XP, członków zespołu nie traktuje się jako całkowicie zamienialny zasób (jak to ma miejsce w metodykach tradycyjnych), ale jako osoby świadome, zaangażowane i współodpowiedzialne za cel projektu. Wspólna praca nad projektem to z jednej strony odpowiednie pomieszczenie (zwykle jedno dla całego zespołu typu otwartego - „open space") i komunikacja w zespole, a z drugiej – odpowiednie zaangażowanie wszystkich członków zespołu, w tym prawdziwego klienta. Ten warunek (a właściwie praktyka) jest jednym z najtrudniej spełnianych.

Zbiór praktyk Programowania ekstremalnego jest ustawicznie rozbudowywany. Poszczególni praktycy i autorzy książek dodają nowe. Pojawiają się w nim praktyki zapożyczone z innych metodyk zwinnych oraz z innych dobrych praktyk wytwarzania oprogramowania. Są to takie praktyki jak: retrospekcje, ciągłość zespołu, krótkie spotkania robocze, demonstracja iteracji, kontrola wersji, dokumentowanie, wizja, szacowanie, optymalizacja wydajności, negocjowany zakres prac czy opłata za użytkowanie. Metodyka XP jest rozwijana i modyfikowana w miarę pojawiający się doświadczeń ze stosowania praktyk zwinnych.

1.4. SCRUM W WYTWARZANIU OPROGRAMOWANIA

Metodyka Scrum (pol. *Młyn)* jest drugą co do popularności metodyką zwinną. Swoją nazwę wzięła od nazwy formacji robionej przez graczy w rugby nad piłką w czasie wznowienia gry (Koszlajda, 2010). Scrum jest kompletną (w odróżnieniu od XP) iteracyjną i przyrostową metodyką realizacji projektów informatycznych. Daje zespołom konkretny oraz spójny zestaw metod i narzędzi do prowadzenia projektów, pozwalając jednocześnie na dużą swobodę w ich doborze i dostosowaniu do warunków firmy bądź tez przedsięwzięcia (Shwaber, 2004). Metodyka Scrum dotyczy wyłącznie fazy wytwarzania oprogramowania i nie obejmuje faz początkowych (opracowanie wizji produktu i strategii jego rozwoju) oraz takich elementów jak zarządzanie kontraktem. Scrum jest metodyką koncentrującą się na procesie. Jest więc jest niejako przeciwieństwem XP. Scrum posiada wiele cech metodyk tradycyjnych, takich jak podejście proceduralne, istotnie sformalizowane oraz prowadzące do konkretnych rozwiązań. Tym niemniej w metodyce Scrum wykorzystywane są niektóre praktyki XP. W wielu projektach „standardowy" Scrum jest rozszerzany przez zespoły o wybrane praktyki XP, potrzebne w danym projekcie i w danych warunkach (Koszlajda, 2010).

Zespół w Scrum składa się z osób, które pełnią w nim następujące role:

- Właściciel produktu,
- Mistrz,
- Członek zespołu,
- Interesariusz.

W zależności od roli może ją pełnić jedna lub więcej osób.

Właściciel produktu (ang. *Product Owner*) jest przedstawicielem klienta lub sponsorem projektu (rozumiany analogicznie jak w metodykach tradycyjnych – rys. 1.2). Jest on odpowiedzialny za kwestie biznesowe projektu: wizję, cele, efektywność itd. Powinien mieć możliwość zatwierdzania (lub jego spowodowania) zmian w planach i podejmowania (lub powodowania podejmowania) wiążących

decyzji dotyczących funkcjonalności tworzonego oprogramowania (analogicznie do sponsora). Tworzy (lub powoduje by powstały) wizję projektu, listę wymagań do systemu – tzw. **rejestr produktu** (ang. *Product Backlog*), listę celów zwrotu inwestycji oraz plany wydań. Do tworzenia listy wymagań właściciel produktu używa zwykle techniki opowieści użytkownika, eposów (tj. dużych opowieści użytkownika) oraz tematów (tj. zbiorów powiązanych ze sobą opowieści). Każdy z elementów rejestru produktu powinien być zgrubnie oszacowany (np. w skali: S, M, L, XL, XXL). Rejestr jest „żywym dokumentem", modyfikowanym w miarę potrzeb przez Właściciela produktu. On też jest odpowiedzialny za uporządkowanie (zwykle poprzez priorytetyzację) wymagań. Właściciel produktu akceptuje lub odrzuca rezultaty prac.

Mistrz (ang. *ScrumMaster*), zwany niekiedy liderem zespołu, to osoba odpowiedzialna z kontrolę procesu realizacji projektu. Pełni również rolę coachera. Do jej podstawowych zadań należy nadzorowanie poprawności realizacji procesu Scrum, nauczanie metodyki i wspomaganie jej prawidłowego użycia przez wszystkich uczestników projektu, implementację Scrum w taki sposób by mieściła się ona w organizacji i kulturze firmy, a także spowodowanie by osoby zaangażowane w projekt postępowały zgodnie z zasadami i praktykami Scrum.

Członek zespołu jest osobą realizującą razem z innymi rozwój funkcjonalności. W Scrum zespoły same się organizują, same się zarządzają i pracują każdy nad swoją funkcjonalnością. Członkowie zespołu w Scrum ponoszą grupową odpowiedzialność za rezultat projektu. Są oni wszyscy równoprawni, tj. nie ma w zespole hierarchii. Typowy zespół Scrum składa się z od 5. do 9. osób (7±2), które ze względu na interdyscyplinarny zakres ich pracy muszą mieć dość duże doświadczenie i posiadać różnorodne kompetencje (muszą być jednocześnie analitykami, projektantami, programistami, inżynierami systemowymi i technicznymi, testerami itd.). Zbyt mały zespół (<5) zmniejsza produktywność i mogą pojawić się w nim braki kompetencji, a zbyt duży (>9) zwiększa problemy z koordynacją prac, wprowadza bałagan itd. Zespół powinien być silnie zmotywowany do osiągnięcia celu projektu. Zespół przedstawia rezultaty prac Właścicielowi produktu.

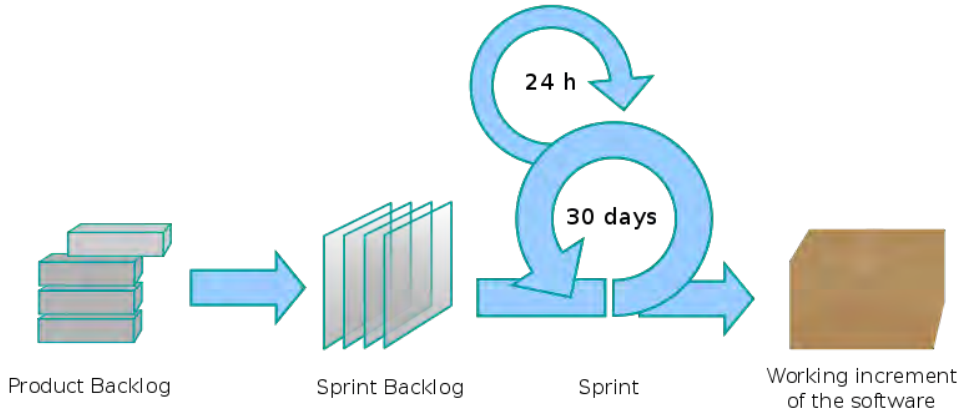
Interesariusze (ang. *Stakeholders*) – są to osoby zainteresowane projektem, ale niezaangażowane (w konsekwencji nie są one odpowiedzialne za rezultat prac) w sposób bezpośredni i aktywny w prace. Są one bardziej obserwatorami niźli

wykonawcami. Osoby te mogą być często pomocne w realizacji projektu, mogą uczestniczyć np. w zebraniach zespołu. Nie mogą one natomiast decydować, bo *de facto* nie ponoszą żadnej odpowiedzialności.

Scrum bardzo wyraźnie wydziela grupy osób zaangażowanych w projekt od osób niezaangażowanych, tzn. interesantów, zwanych żartobliwie kurczakami (ang. *Chickens*) w przeciwieństwie do świń (ang. *Pigs*), czyli osób podejmujących zobowiązania projektowe. Rozgraniczenie to związane jest z zasadą jasnego rozdzieleniem ról i odpowiedzialności w projekcie, unikaniem niejasności decyzyjnej lub ponoszenia odpowiedzialności za nie swoje decyzje, a poprzez to zwiększenie produktywności zespołów.

Cykl życia projektu w metodyce Scrum (rys. 1.7) składa się z szybkich i intensywnych cyklów wytwórczych zwanych **sprintami**. Na początku każdego sprintu tworzony jest na podstawie zaległości produktowych (tj. niezrealizowanych jeszcze elementów rejestru produktu) tworzona jest lista zadań sprintu, zwana **rejestrem sprintu** (ang. *Sprint Backlog*). Zadania te są realizowane w trakcie sprintu. Rejestr zadań nie jest zmieniany w trakcie sprintu.

Sprint wykonywany jest w następujących po sobie cyklach dziennych. Elementem obowiązkowym każdego dnia jest **poranne spotkanie sprintu** (ang. *Sprint Meeting*), zwane też **codziennym młynem** (ang. *Daily Scrum)*. Spotkanie to jest krótkie (do 15 minut) i realizowane zwykle na stojąco codziennie w tym samym miejscu. Codzienny młyn jest poświęcony omówieniu zadań zrealizowanych poprzedniego dnia, problemów pojawiających się przy ich realizacji oraz zadań do realizacji danego dnia. Sprint kończy sie dostarczeniem działającego oprogramowania, zawierającego nowe funkcjonalności (ang. *Working Increment of the Software*) oraz jego prezentacją na **spotkaniu inspekcyjnym** (ang. *Sprint Review*). Ostatnim działaniem w sprincie jest **retrospekcja sprintu** (ang. *Sprint Retrospective*), realizowana na spotkaniu retrospektywnym (ang. *Postmortem Meeting*).



Rys. 1.7. Schemat metodyki Scrum
Źródło: (http://pl.wikipedia.org/wiki/Scrum)

Sprinty trwają stosunkowo krótko (od dwóch do czterech tygodni) i nie muszą mieć stałej długości w trakcie trwania projektu. Czas trwania sprintu ustala zespół. Metodyka zaleca co prawda stałe długości sprintu, ale zwykle sprinty na początku projektu (wytworzenie pierwszej wersji prototypu systemu) trwają dłużej (np. dwa miesiące), a pod koniec, gdy zachodzi potrzeba biznesowa wniesienia zmian do już działającego produktu, sprint może być bardzo krótki (np. jeden tydzień). Sprinty zwykle następują jeden po drugim, ale dopuszczalne są również przerwy pomiędzy nimi. Przerwy mogą wynikać z czynników niezależnych od zespołu (np. braku informacji od klienta) lub być przez niego zaplanowane (np. okres urlopowy, święta). W rezultacie sprintu zespół dostarcza działający, przetestowany prototyp systemu, zawierający nowe funkcjonalności opracowane w czasie jego trwania.

Sprinty zaczynają się po tym jak zostanie opracowany przez Właściciela produktu rejestr produktu, czyli lista wymagań do oprogramowania. Lista ta powinna zostać uporządkowana z wykorzystaniem wartości biznesowych funkcjonalności.

Poszczególne sprinty są planowane na specjalnym **spotkaniu planistycznym sprintu** (ang. *Sprint Planning Meeting*) trwającym nie dłużej niż cztery godziny. Takie ograniczenia czasowe (ang. *Timebox*) są charakterystyczne dla Scrum i dotyczą większości zadań, związanych z organizacja procesu wytwórczego. W trakcie

spotkania zespół, we współpracy z Właścicielem produktu, ustala czas trwania sprintu i opracowuje rejestr sprintu. Jest to lista zadań do zrealizowania w danym sprincie. Powstaje ona na podstawie jednego lub więcej wymagań do systemu o najwyższym priorytecie w aktualnym rejestrze produktu. Wymaganie takie jest zdefiniowane na poziomie biznesowym i zwykle wymaga dekompozycji na mniejsze elementy - zadania do wykonania. Zadania powinny być tak zdefiniowane, by można było oszacować koszty ich realizacji i sprawdzić fakt ich realizacji. Zaleca się by nie były zbyt duże, tj. by pracochłonność ich wykonania nie przewyższała 2. roboczodni. Każde zadanie powinno zostać oszacowane pod kątem kosztów jego realizacji. Szacunki te wykonywane są z wykorzystaniem **metryk czasowych** (ang. *Time-Based Estimation*) lub **względnych** (ang. *Relative Estimation*). Metryki i zasady ich użycia są opisane w następnym rozdziale. Na podstawie potencjalnych możliwości zespołu (tj. budżetu czasu jego pracy), kosztów realizacji poszczególnych zadań tworzy się rejestr zadań sprintu. Rejestrem tym, po jego ustaleniu na spotkaniu planowania sprintu, zarządza zespół (np. ustala kolejność realizacji, przydziela zadania do programistów itd.). Rejestr jest niezmienny (zamrożony) do końca sprintu. W szczególnych przypadkach sprint może być przerwany. Przykład rejestru przedstawia rys. 1.8.

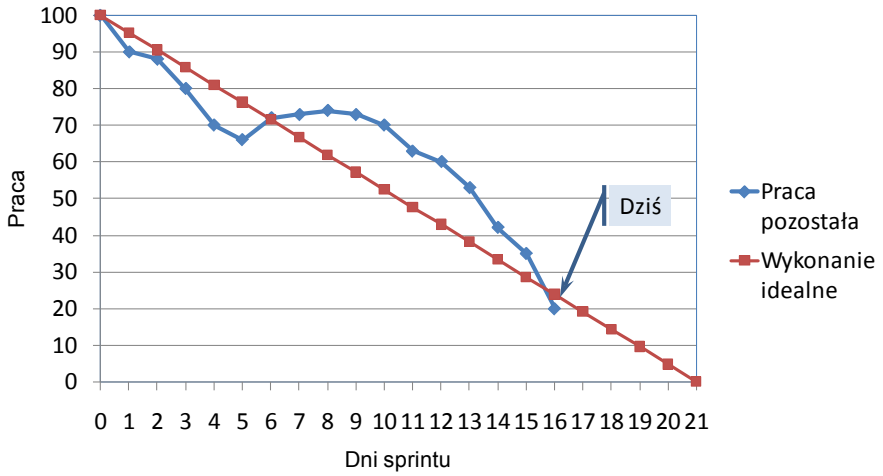
Element funkcjonalności	Zadanie	Właściciel	Estymacja pracy
Logowanie się do systemu	Skonfigurowanie i przetestowanie bazy danych do przechowywania identyfikatorów i haseł	Jan	2 h
	Implementacja formularza logowania wraz z komunikatami o błędach	Marek	4 h
	Implementacja zmiany hasła jednorazowego użytkownika	Marek	6 h
Wprowadzanie danych o użytkownikach z nadawaniem haseł jednorazowych	Implementacja formularza wprowadzania danych o użytkownikach	Stefan	4 h
	Budowa i testowanie algorytmu nadawania haseł jednorazowych	Ela	2 h
	Implementacja zabezpieczenia przed powtórnym wprowadzeniem danych użytkownika	Zosia	8h

Rys. 1.8. Fragment rejestru zadań sprintu - przykład

Źródło: opracowanie własne

W pracy nad produktem wykorzystywane są praktyki XP (projektowanie, kodowanie, refaktoryzacja, testowanie, integracja itd.).

W trakcie wspomnianego już codziennego spotkania (czyli Młynu) rozpoczynającego dzień pracy, Mistrz aktualizuje **wykres wypalania** (ang. *Burndown Chart*), zwany także wykresem zaległości. Wizualizuje on (rys. 1.9) postępy prac poprzez pokazanie dzień po dniu (od początku sprintu do dnia bieżącego) ile pracy do wykonania jeszcze pozostało (praca pozostała - rys. 1.9). Na osi poziomej wykresu odkładane są dni sprintu, a na osi pionowej praca do wykonania w jednostkach szacowania kosztów realizacji zadań. Idealna realizacja projektu przewiduje równomierne obciążenie zespołu, a więc i liniowe „wypalanie" zadań. Taką sytuację przedstawia linia wykonania idealnego na rys. 1.9. Linia ta (a właściwie jej nachylenie) obrazuje także średnią planowaną wydajność zespołu (ang. *Team Velocity*). Wykonywanie zadań dzień po dniu zmniejsza ilość pracy, która pozostaje do końca sprintu. Możliwy jest przyrost pracy do wykonania (np. na rys. 1.9 w okresie od 6. do 8. dnia sprintu). Sytuacja taka powstaje w wyniku konieczności ponownego wykonania zadań, pojawienia się nowych zadań w rejestrze sprintu lub zmiany ich oszacowania w trakcie realizacji.



Rys. 1.9. Wykres wypalania sprintu - przykład

Źródło: opracowanie własne

Na podstawie wykresu wypalania szybko można ocenić i kontrolować postępy prac w projekcie. Jeśli bowiem linia pracy pozostałej do wykonania po danym dniu (rys. 1.9 - dzień 1, 2 do 5) jest poniżej linii wykonania idealnego to projekt jest realizowany z wyprzedzeniem harmonogramu, w przeciwnym przypadku (rys. 1.9 - dni od 7 do 15) - projekt ma opóźnienie.

Jedną z technik Scrum (ale nie tylko, bo inne metodyki zwinne też się nią posługują), zapewniającą prostotę komunikacji w zespole i z otoczeniem jest tablica wizualizacji projektu - rys. 1.10. Jest ona podstawowym narzędziem do bieżącego odwzorowywania stanu oraz zarządzania sprintami. Przedstawia ona rejestry produktu i zadań, poszczególne sprinty oraz stan zadań (już rozpoczęte i nie-, zakończone i planowane do zakończenia) a także dodatkowe informacje o projekcie, zespole, rezultatach kontroli itd.

Rys. 1.10. Tablica wizualizująca przebieg projektu w Scrum - przykład
Źródło: http://www.xqa.com.ar/visualmanagement/

W trakcie codziennego spotkania członkowie zespołu komunikują problemy z realizacją zadań. Niekiedy problemy te całkowicie wstrzymują pracę (np. brak ustalenia zawartości formularza), a zespół nie może sobie z tym poradzić

samodzielnie. Taki problem nazywa sie blokadą. **Rejestr blokad** (ang. *Impediments Backlog*) prowadzi Mistrz, ustala z zespołem ich priorytety, używając dostępnych mu mechanizmów usuwa je (lub powoduje usunięcie) oraz raportuje zespołowi ich stan.

W ostatni dzień sprintu odbywają się dwa spotkania - każde trwające nie więcej niż 4 godziny. Pierwsze z nich jest **inspekcją rezultatów** pracy w sprincie (ang. *Sprint Review*). W jego trakcie wszystkim zaangażowanym osobom przedstawiany jest rezultat sprintu (funkcjonujące oprogramowanie, wydanie), a Właściciel produktu stwierdza (lub nie) wykonanie zadań sprintu. Zespół omawia na nim zaistniałe problemy (oraz zadania zakończone sukcesem) i metody ich rozwiązania. Właściciel produktu omawia stan rejestru produktu i jego modyfikacji. Zadania z rejestru sprintu, które nie zostały zrealizowane w danym sprincie (a może sie to zdarzyć bo sprint ma ustalony czas trwania) przechodzą za zgodą Właściciela produktu do nowego sprintu. W trakcie **spotkania retrospektywnego**, które kończy ostatni dzień sprintu i trwa nie dłużej niż 4 godziny, zespół z Mistrzem dokonuje analizy pozytywnych i negatywnych elementów sprintu. W przypadku zidentyfikowania problemów planowane są działania usprawniające i naprawcze. Spotkanie retrospektywne jest, znaną z metodyk tradycyjnych, metodą pozyskiwania wiedzy o realizacji projektu oraz refleksją zespołu nad przebiegiem prac. W Scrum jest ono realizowane wielokrotnie w trakcie projektu, co umożliwia wykorzystanie jego wyników jeszcze w danym projekcie, tj. już w kolejnym sprincie.

W szczególnych przypadkach Właściciel produktu może przerwać sprint (ang. *Abnormal Termination*). Dzieje się to na podstawie informacji pochodzącej od zespołu (np. istotny problem techniczny, niemożliwy do rozwiązania w zaplanowanym sprincie), Mistrza (np. brak realizacji metod Scrum), bądź też od zamawiającego (np. zmieniły sie warunki zewnętrzne i realizacja zadań w sprincie nie ma dalej sensu).

Scrum jest metodyką zwinną, która próbuje wprowadzić pewne formalizmy porządkujące proces tworzenia oprogramowania. Są to: struktura procesowa, ścisły podział obowiązków pomiędzy uczestników projektu oraz stosowanie sformalizowanych technik nadzoru nad projektem. Scrum narzuca dużą samodyscyplinę zespołom, na którą składają się codzienne spotkania czy też prezentacja rezultatów sprintu. Jednocześnie Scrum zachowuje cały szereg elementów zwinnych.

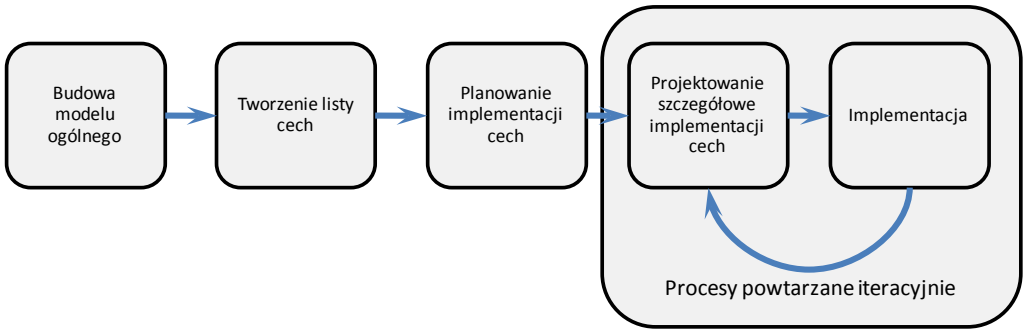
1.5. METODYKA FDD

Metodyka FDD (ang. *Feature Driven Development*) jest ukierunkowana na wspomaganie zarządzania projektami tworzenia oprogramowania w typowych fazach i krótkich iteracjach, połączonych z dostarczaniem gotowego produktu klientowi. Metodyka zapewnia jednocześnie dostęp do wiarygodnych informacji o projekcie dla wszystkich jego uczestników (Coad, 1999). FDD posiada cały szereg cech wspólnych dla metodyk zwinnych, ale wprowadza więcej formalizmu. Przejawia się on miedzy innymi w strukturyzacji dużych zespołów, wiarygodnym śledzeniu postępu prac czy częstych inspekcjach kodu. Niektóre jej cechy, takie jak przypisanie kodu do j ego właściciela, tj. do konkretnego programisty (cecha: indywidualna odpowiedzialność za rezultat pracy) nie są zgodne z innymi metodykami zwinnymi. FDD umożliwia (Palmer, 2002) realizację projektów dużych (do 500 programistów), krytycznych w aspekcie funkcjonalności, czasu i budżetu przez programistów o zróżnicowanych kwalifikacjach (w tym, i niskich). Wprowadzając formalizmy i procesowe podejście, metodyka FDD umożliwia utrzymanie projektu pod kontrolą przy wykorzystaniu w dużej jego części podejścia zwinnego. Struktura organizacyjna projektu w FDD jest silnie zhierarchizowana. Wyróżnia sie w niej następujące główne role uczestników:

- **Kierownik projektu** - osoba odpowiedzialna za całość projektu. Zarządza ona zakresem, harmonogramem i zasobami (identycznie jak w metodykach tradycyjnych).
- **Główny architekt** - odpowiedzialny za architekturę oprogramowania, za ogólny projekt systemu i jego spójność.
- **Eksperci dziedzinowi** - osoby, które przekazują programistom wiedzę na temat wymagań funkcjonalnych do opr ogramowania. Są to przedstawiciele klienta, użytkownicy, analitycy biznesowi, osoby merytoryczne w danej dziedzinie.
- **Kierownik programistów** - osoba zarządzająca pracą wielu zespołów programistów. Koordynuje ona ich działania, rozwiązuje konflikty o zasoby itd.
- **Główni programiści** - doświadczeni programiści pełniący rolę liderów niewielkich (3-6 osobowych) zespołów realizujących określonych fragment systemu. Poza bieżącym zarządzaniem zespołem, biorą oni udział w analizie wymagań oraz projektowaniu rozwiązania (razem z Głównym architektem).

- **Właściciele klas** (tj. fragmentów kodu) - programiści rozwijający poszczególne cechy systemu. Pod rozwojem rozumie się w FDD projektowanie szczegółowe, kodowanie, testowanie oraz dokumentowanie poprogramowania.

Metodyka FDD zakłada realizację projektu w cyklu, składającym się z pięciu faz, przy czym dwie ostatnie fazy są powtarzane wielokrotnie, iteracyjnie (rys. 1.11). Jest ona zatem połączeniem kaskadowej (szeregowej) struktury zadaniowej znanej z metodyk tradycyjnych z iteracyjną, preferowaną przez metodyki zwinne.



Rys. 1.11. Cykl życia projektu w metodyce FDD
Źródło: opracowanie własne na podstawie (Palmer, 2002)

W trakcie planowania i realizacji projektu w metodyce FDD wykorzystuje się pojęcie **cechy** (ang. *Feature*). Jest to mały fragment funkcjonalności, posiadający jako całość wartość dla klienta. Cecha definiowana jest zwykle w postaci: <akcja> <rezultat> <obiekt> (np. „Wydruk listy studentów" lub „Weryfikacja hasła użytkownika"). Cecha powinna być definiowalna przez klienta i dość mała, tak by czas jej implementacji przy pomocy zespołu programistów nie przekraczał 2. tygodni. W przypadku pojawienie się bardziej rozbudowanej cechy należy ją zdekomponować na prostsze. Cechy definiują zatem wymagania funkcjonalne do oprogramowania w prosty sposób. Cechy mogą być biznesowo powiązane ze sobą i tworzyć **zbiory cech** (ang. *Features Sets*). Zbiór cech może być realizowany w ramach jednego komponentu i może być implementowany iteracyjnie. W większych projektach metodyka FDD wprowadza trzeci poziom dekompozycji wymagań funkcjonalnych,

a mianowicie **obszar przedmiotowy** (ang. *Subject Area*). Jest on odp owiednikiem podsystemów lub pakietów w UML (ang. *Unified Modeling Language*).

Każda faza w projekcie definiowana jest przy pomocy klasycznego układu: wejście-proces-wyjście. FDD definiuje fazy poprzez określenie kryteriów wejściowych (co trzeba mieć?), zadań do realizacji (struktura procesu wraz z odpowiedzialnym wykonawcą) i kryteriów wyjściowych (tj. warunków zakończenia fazy). Fazy dzielone są na zadania cząstkowe.

Metodyka FDD wyróżnia następujące fazy (rys. 1.11):

- budowa modelu ogólnego,
- tworzenie listy cech,
- planowanie implementacji cechy,
- projektowanie szczegółowe implementacji cechy,
- implementacja.

Budowa modelu ogólnego (ang. *Develop Overall Model*). Faza ta jest poświęcona zorganizowaniu zespołu projektowego, analizie ogólnej wymagań do systemu, podziałowi obszaru projektowego na do meny i przygotowaniu modeli obiektowych domen w małych podgrupach, wypracowaniu i zatwierdzeniu ogólnego modelu systemu oraz jego udokumentowaniu. Wypracowany na tym etapie model systemu dostarcza wiedzy o ogólnych wymagania do systemu i jest „przewodnikiem" w następnych fazach.

Tworzenie listy cech (ang. *Build Feature List*). Na podstawie modelu ogólnego oraz wiedzy pozyskanej przez zespół w poprzedniej fazie tworzona jest lista cech tworzonego systemu (ang. *Feature List*). Stosuje się tu podejście dekompozycji funkcjonalnej (znane doskonale z inżynierii oprogramowania) wyodrębniając obszary, dzieląc je na zbiory cech i poszczególne cechy. W rezultacie powstaje uporządkowana, hierarchiczna lista cech oprogramowania do realizacji.

Planowanie implementacji cech (ang. *Plan by Feature*). Głównym celem tej fazy jest stworzenie planu, który definiuje kolejność realizacji poszczególnych cech. Kolejność ta wynika przede wszystkim z priorytetów klienta oraz technologicznej zależności pomiędzy cechami (np. „Wydruk listy studentów" po „Rejestracja danych osobowych studentów"), a także złożoności implementacyjnej i możliwości zespołów.

Plan implementacji ma być tak skonstruowany by jego realizacja była mierzalna zewnętrznie. W związku z czym wprowadza się do niego punkty kontrolne - kamienie milowe (ang. *Milestones*). Plan implementacji określa momenty realizacji każdego zbioru cech, utworzonego w poprzedniej fazie. Każdy zbiór cech przydzielany jest do zespołu (ang. *Feature Team*) i głównego programisty. Po tym, cechy są przypisywane w ramach zespołu do poszczególnych programistów - właścicieli klas.

Projektowanie szczegółowe implementacji cech (ang. *Design by Feature*). Faza ta polega na kolejnym przeglądzie dziedziny problemu i dokumentów dla każdego, wyodrębnionego w poprzedniej fazie zbioru cech, stworzenie diagramów sekwencji (ang. *Sequece Diagrams*) dla cech, uszczegółowienie modelu obiektowego (stworzenie i uszczegółowienie modelu klas) i przeprowadzenie końcowej inspekcji projektu (kontrola poprawności projektu szczegółowego) pod kątem jego poprawności. W trakcie tej fazy możliwa jest także modyfikacja modelu ogólnego, która może doprowadzić do modyfikacji zbiorów cech i planów ich implementacji.

Implementacja (ang. *Build by Feature*). Faza ta zaczyna się po pozytywnym ukończeniu inspekcji projektu szczegółowego. W jej trakcie programiści (właściciele klas) implementują funkcjonalności zaprojektowane w fazie poprzedniej. Po implementacji, wykonywane są testy jednostkowe i inspekcja (przegląd) kodu. Pozytywne ich ukończenie prowadzi do zintegrowania nowego kodu (nowych cech - funkcji) z oprogramowaniem, które było dostarczone klientowi. Działania integracyjne są realizowane regularnie. Umożliwia to wcześniejsze wykrywanie błędów integracyjnych oraz dysponowanie gotowym produktem, który może być przekazany klientowi w dowolnym momencie.

W celu raportowania, śledzenia i kontroli realizacji poszczególnych cech w trakcie iteracji (a są one niezbyt złożone, a więc generują proste zadania implementacyjne) w FDD zdefiniowane zostało sześć punktów kontrolnych (ang. *Milestones*). Punkty kontrolne obejmują dwie ostatnie fazy, tj. Projektowanie szczegółowe i Implementację. Trzy pierwsze punkty kontrolne dotyczą fazy Projektowanie szczegółowe implementacji cech, a pozostałe do kolejnej. Powinny być one osiągane dla każdej cechy jeden po drugim sekwencyjnie i sprawdzane w trakcie realizacji oprogramowania cechy. W celu ujednoliconego ustalania postępów prac do poszczególnych punktów kontrolnych przypisane zostały procentowe poziomy

zaawansowania prac - tab. 1.1. Projektowanie szczegółowe implementacji cechy kosztuje zatem 44% pracy w danej iteracji, a 56% - implementacja.

Tabela 1.1. Punkty kontrolne w metodyce FDD i poziom zaawansowania prac

	Zaawansowanie	**Faza**
Przegląd dziedziny	1%	Projektowanie szczegółowe
Projekt szczegółowy	40%	Projektowanie szczegółowe
Inspekcja projektu	3%	Projektowanie szczegółowe
Kod programu	45%	Implementacja
Inspekcja kodu	10%	Implementacja
Przekazanie do integracji	1%	Implementacja

Źródło: (Palmer, 2002)

Podobnie jak inne metodyki zwinne, FDD wykorzystuje fundamentalne dla niej praktyki, z których znaczna część pochodzi z Manifestu zwinności. Do podstawowych praktyk FDD zalicza się (Palmer, 2002):

- **Koncentracja na wymaganiach klienta**. Przejawia się ona w procesie tworzenia projektu architektury systemu oraz listy cech oprogramowania i ukierunkowaniu go na realne potrzeby klienta. W procesy te zaangażowani są przedstawiciele klienta, co zwiększa koncentrację metodyki na potrzebach klienta. Każda kolejna iteracja dostarcza klientowi przyrostu oprogramowania istotnego z punktu jego interesów biznesowych. Ponadto metodyka FDD umożliwia także klientowi realną kontrolę postępów prac. Wykorzystuje się do tego listy cech i punktów kontrolnych procesu ich realizacji.
- **Architektura systemu sterowana modelem**. W FDD szczególne znaczenie przypisywane jest modelowi ogólnemu systemu, opracowywanemu na początku projektu z wykorzystaniem technik obiektowych. Można ją zatem zaliczyć do metodyk wykorzystujących model (ang. *Model Driven Architecture, MDA*). Modelowanie systemu ma cały szereg zalet: umożliwia lepsze zrozumienie całości systemu, czyni go bardziej spójnym, umożliwia opracowania planu implementacji systemu oraz wspomaga wprowadzanie modyfikacji. FDD dopuszcza bowiem

modyfikację modelu w trakcie iteracji. W zależności od konkretnych warunków realizacji projektu (np. wymagania zamawiającego by model po zatwierdzeniu był niemodyfikowalny) model ogólny może być opracowany w 100%, zatwierdzony i ustabilizowany. Wykorzystuje się wówczas cykl kaskadowy w FDD. W innych przypadkach, model nie musi być kompletny (np. definiować tylko 60% wymagań). Jego dodefiniowywanie może zostać pozostawione na iteracyjne fazy wytwórcze. Takie podejście czyni FDD metodyką zwinną. Możliwe są oczywiście wszystkie warianty pośrednie. Podobnie (tj. pełna lub tylko częściowa stabilizacja) sprawa może się mieć z rezultatem fazy Planowania implementacji cech, czyli planem wytwarzania.

- **Krótkie iteracje**. Częste dostarczanie funkcjonującego i zintegrowanego kodu, zawierającego coraz to nowe funkcjonalności są elementem FDD. Implementacja cechy w czasie nie większym niż 2 tygodnie czyni FDD niezwykle zwinną. Umożliwia dostarczanie w kolejnych iteracjach najbardziej potrzebnych (o najwyższym priorytecie klienta) funkcjonalności.
- **Indywidualna odpowiedzialność za klasę**. Praktyka ta odróżnia FDD od innych metodyk zwinnych. Ma ona za zadanie zwiększenie jakości kodu oraz zmniejszenie kosztów jego modyfikacji. Programista doskonale zna kod i identyfikuje się z nim.
- **Inspekcje**. FDD dużą rolę przywiązuje do inspekcji projektu, jako mechanizmowi wykrywania błędów, ale też i przepływu wiedzy o systemie oraz standaryzacji rozwiązań.
- **Regularne budowanie produktu**. Każda iteracja FDD dostarcza gotowego do użycia oprogramowania, powiększonego o funkcjonalność cechy. Częsta integracja oprogramowania umożliwia wczesne wykrycie błędów oraz daje możliwość automatyzacji testów regresywnych. Jest to praktyka wynikająca bezpośrednio z Manifestu zwinności.
- **Zarządzanie konfiguracją**. W FDD pod tym pojęciem rozumie się nie tylko przechowywanie (najlepiej w systemie kontroli wersji) kodu źródłowego, ale całej powstającej dokumentacji (w tym wymagań klienta) i jej zmian. Ułatwia to śledzenie zmian i usprawnia zarządzanie projektem.

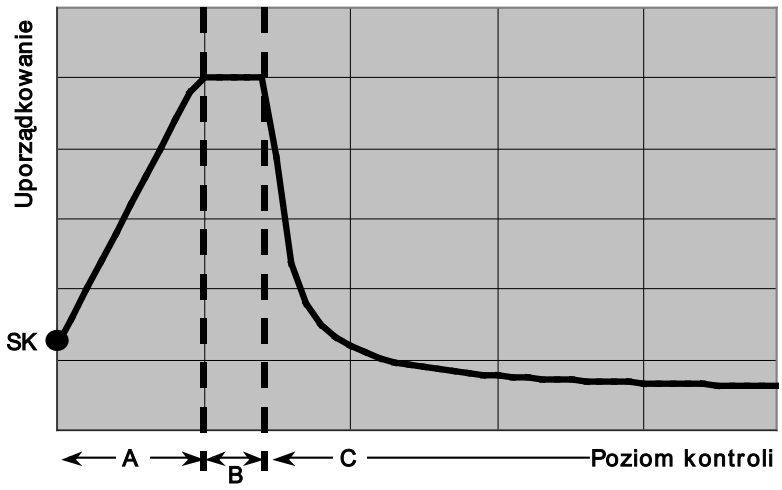
- **Raportowanie i widoczność wyników**. FDD dostarcza narzędzi do usprawnienia procesu monitorowania przebiegu projektu. Śledzenie, wykorzystujące punkty kontrolne oraz listy cech, jest proste do realizacji, raportowania i wizualizacji.

FDD łączy cechy metodyk zwinnych z tradycyjnymi, wprowadzając formalizmy i dając możliwości nadzoru nad projektem (i to zarówno w fazie wstępnej, jak i wykonawczej). Narzędzia i stosowane w FDD praktyki zwiększają poziom kontroli do rozsądnego z punktu widzenia klienta i zespołu wykonawczego.

1.6. WDRAŻANIE PRAKTYK ZWINNYCH

Metodyki zwinne zakładają, że zespoły realizujące projekt pod okiem bardziej mentora niż kierownika (np. Mistrz w Scrum czy Coacher z XP). Jednym z założeń metodyk zwinnych jest to by zespoły same się organizowały, zarządzały pracą (oczywiście po wstępnych ustaleniach - np. po sformułowaniu Rejestru sprintu w Scrum) i same kontrolowały jej wykonanie (np. programowanie parami w XP czy też inspekcje kodu w FDD). Założenie to jest zgodne z doświadczeniem pracy wielu zespołów programistów. Zdyscyplinowani (samo-), zdeterminowani (samo-) i odpowiednio zdopingowani (samo-) członkowie zespołów, realizujący ciekawe prace w krótkich cyklach nie muszą być w ogóle kontrolowani w trakcie tych cykli.

Samoorganizacja zespołu powoduje, że w projektach, pomimo braku jakiekolwiek nawet kontroli zewnętrznej, jest pewien poziom uporządkowania (wartość SK na rys. 1.12). Poziom ten zwiększa się poprzez podkreślenie (często stosowane w metodykach lekkich) silnego związku uczuciowego (twórca, artysta, właściciel itp.) członka zespołu z rezultatami jego pracy - kodem (np. całością oprogramowania w XP) lub jego fragmentem (np. klasa w FDD).



Rys. 1.12. Realna zależność pomiędzy poziomem kontroli a uporządkowaniem projektów
Źródło: opracowanie własne (Miłosz, 2006) na podstawie (Agile, 2011)

Możliwe jest podniesienie poziomu uporządkowania w projekcie ponad wartość SK (pol. ***S**amo**k**ontrola*), poprzez wprowadzanie elementów kontroli, a więc procesów planowania, wykonania, rozliczenia, inspekcji, realizowanych przez osoby do tego desygnowane. Takie działania są efektywne, ale tylko do pewnego momentu – obszar A na rys. 1.12. Po przekroczeniu pewnego progu, wprowadzanie kolejnych procedur kontroli nie powoduje wzrostu uporządkowania (obszar B na rys. 1.12). Dalsze wzmaganie procesów kontroli powoduje wręcz spadek porządku w projekcie (obszar C). Nadmiar procedur kontrolnych doprowadza bowiem do sytuacji, że osoby je realizujące są przeciążone, zbierane (lub raportowane) dane kontrolne są niedokładne, pojawiają się przekłamania lub wręcz fałszerstwa. Pojawia się również zjawisko „przekleństwa ilości" - danych o projekcie jest zbyt dużo, by je można było racjonalnie wykorzystać. Projekt na „papierze" zaczyna wyglądać całkiem inaczej niźli w rzeczywistości. W raportach zwykle jest „idealnie", zgodnie z oczekiwaniami służb kontrolnych (Miłosz, 2006), a w praktyce - różnie.

W ostatnim czasie stało się modne wdrażanie metodyk zwinnych w firmach produkujących oprogramowanie. Zwinność traktowana jest jako panaceum na wszelkie bolączki, związane z wytwarzaniem oprogramowania. Proces zmiany

metodyki zarządzania projektami w skali całej firmy jest dużym wyzwaniem organizacyjnym. Sprzyja ono popełnianiu całego szeregu pomyłek i błędów, które z kolei zwiększają koszty zmiany oraz doprowadzają do wzrostu problemów na etapie przejściowym. Gurses L. zidentyfikował i sklasyfikował 10 typowych błędów we wdrożeniu metodyk lekkich (Gurses, 2006):

1. Jednorazowe przejście całej firmy na nową metodykę.
2. Szybkie przejście by szybko uzyskać rezultaty.
3. Zapominanie o kulturze korporacyjnej.
4. Błędna identyfikacja Sponsora projektu.
5. Złe przypisanie ról w zespole.
6. Brak planowania.
7. Zbyt szybkie tworzenie zespołów.
8. Natychmiastowe pozbycie się wszelkich narzędzi wspomagających klasyczne zarządzanie projektami i projektowanie systemów.
9. Zły dobór osób kluczowych w zespołach.
10. Nadmierna ewangelizacja nowych metod.

Powyższe błędy wynikają z trzech przyczyn:

A. Proces i sposób wprowadzania zmiany.
B. Właściwość metodyk zarządzania projektami.
C. Właściwości metodyk zwinnych.

Typowe problemy, związane z samym procesem wprowadzaniem zmiany (A) w organizacji pracy w firmie programistycznej, to (Miłosz, 2006):

- jednorazowe przejście (zmasowane) na metodyki zwinne we wszystkich projektach realizowanych w firmie, bez pilotażu, bez uczenia się na błędach, bez możliwości przyzwyczajenia pracowników i klientów do nowych metod pracy (błąd nr 1);
- natychmiastowe przejście do nowych metod bez uwzględnienia stanu zaawansowania dotychczas realizowanych projektów, zbyt szybkie i niewłaściwie przygotowane, bez szczegółowego definiowania ról i zadań, bez posiadania pracowników o wymaganych kompetencjach, bez przeszkolenia pracowników, bez wymaganych zmian w strukturze organizacyjnej firmy (błąd nr 2);

- łamanie dotychczasowej kultury korporacyjnej firmy, tj. istniejących metod, a także działów je realizujących (np. dział planowania, marketingu, rozwoju czy wsparcia klientów) wraz z historycznie ustanowionymi barierami pomiędzy oddziałami, ich wewnętrzną polityką oraz interesami kierownictwa (błąd nr 3);
- szybkie tworzenie zespołów, swoiste co prawda dla metodyk zwinnych, ale doprowadzające do sytuacji braku możliwości właściwego wykorzystania ich członków i ich kompetencji (starych); podejście takie rodzi frustracje, niekiedy brak przydziału zadań do poszczególnych członków zespołu i w konsekwencji doprowadza do powstania złej opinii w firmie na temat metodyk zwinnych (błąd nr 7);
- szybkie zarzucenie dotychczasowych technik oraz narzędzi wspomagania zarządzania projektami i projektowania systemów bez dostatecznie długiego czasu na wprowadzenie tej zmiany (błąd nr 8);
- totalne zanegowanie wartości dotychczas stosowanych (przez wiele lat, zwykle z przynajmniej częściowym sukcesem) metod i ewangelizacja nowości BEZ widocznych jeszcze pozytywnych efektów stosowania nowych metod (błąd nr 10).

Do błędów, związanych z właściwościami metodyk zwinnych (B) i wdrażaniem w praktyce metodyk zarządzania projektami (niezależnie do jakiej grupy się je zalicza: tradycyjnych, czyli formalnych, czy zwinnych), można zaliczyć błędną identyfikację Sponsora (błąd nr 4), zły podział ról w zespole (błąd nr 5) oraz zły wybór osób kluczowych (błąd nr 9). Są to wielokrotnie popełniane, zdiagnozowane „typowe" błędy realizacji projektów zmiany organizacyjnej.

Z powyższego wynika, że właściwie jedynym problemem związanym z wdrożeniem metodyk zwinnych zarządzania projektami w organizacjach są problemy wynikające z zarzuceniem długofalowego planowania projektu. Jest to dość istotny obszar zarządzania projektem, przy czym często związany z podmiotem zamawiającym projekt (klientem zewnętrznym lub wewnętrznym).

Innymi problemami związanymi z wykorzystaniem metodyk zwinnych w firmach, które wynikają z braku kompleksowego (ogólnofirmowego, korporacyjnego) podejścia, są następujące sytuacje:

- Wdrożenie metodyk zwinnych odbywa się często tylko na poziomie zespołu programistów z całkowitym pominięciem jego otoczenia (czyli całej firmy oraz jej

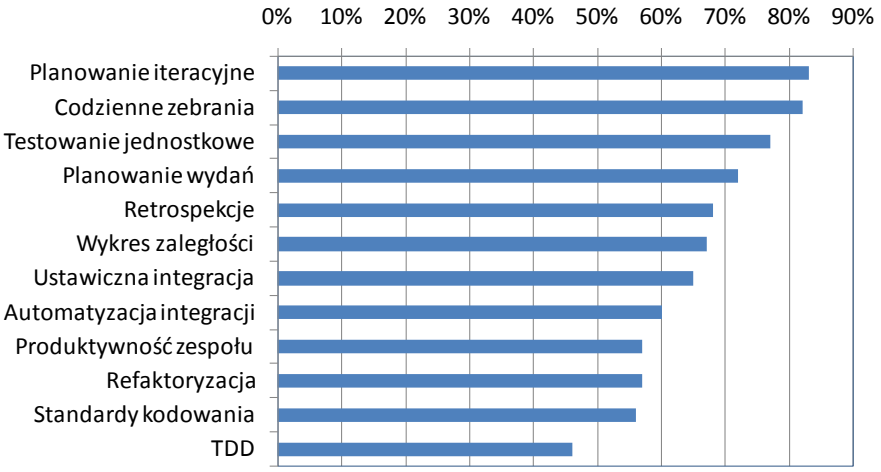
klientów). W konsekwencji inne działy firmy (np. marketing, sprzedaży czy wsparcia) działają „po staremu". Sytuacja taka prowadzi do konieczności walki z niedopasowaniem tradycyjnej działalności tych działów ze zwinnym podejściem zespołu (dotyczy to np. zapisów w zawartych kontraktach, które nie uwzględniają w żaden sposób specyfiki podejścia zwinnego).

- W sytuacjach „podbramkowych" projektu (np. brak czasu czy zasobów), pomimo ogólnej akceptacji metody zwinnych, zespół zmuszany jest do rozwiązywania problemów „po staremu" (np. „szturmując" termin, ograniczając testowanie regresyjne czy też nie uwzględniając pojawiającej się zmiany wymagań klienta do systemu).
- Zespoły, które przestawiły się na metodyki zwinne, często zmuszane są do równoległego stosowania starych metod i narzędzi (np. kontroli wersji, raportowania rezultatów prac itd.) narzuconych przez firmę i wynikających z jej długofalowych zobowiązań (np. umów lub wdrożonego standardu ISO 900x). Prowadzi do nieporozumień i wzrostu nakładów pracy, a także obniża efektywność stosowanych metodyk.
- W przypadku budowy systemu bez konkretnego klienta (na rynek), rolę Właściciela produktu często pełni osoba z innego działu firmy (np. marketingu czy też rozwoju strategicznego), która głównie jest zainteresowana oszczędnościami kosztów, skróceniem czasu, a nie jakością produktu. Osoba ta narzuca zespołowi nierealistyczne oszacowania i terminy. Powoduje to obniżenie jakości produktu, ale też cierpi na tym morale zespołu.
- Osoby pełniące role kierowników projektów, realizowanych uprzednio metodami tradycyjnymi, lekceważąco traktują niektóre metody zwinne (np. kolorowe kartki przyklejane w różnych miejscach, tablica projektu czy spotkania na stojąco przy komputerach) i nagłaśniają je w firmie jako „dziecięce", „niepoważne", „nienaukowe", itp. Zarzucają przy tym, że na takie „zabawy" są marnowane zasoby firmy i doprowadzają one do opóźnień w projektach (które w okresie przejściowym rzeczywiście się zdarzają, ale całkiem z innych przyczyn, niekoniecznie tak widocznych jak tablica z karteczkami).

W 2010 roku były przeprowadzone piąte już badania dotyczące stanu wdrażania praktyk zwinnych (State, 2010). Badaniem objęto 4770 respondentów z 91 krajów świata. Jego rezultaty wskazują na (State, 2010):

- wysoki poziom wiedzy o praktykach zwinnych (tylko 8% wskazało na niski poziom wiedzy), ale też i stosunkowo niski poziom doświadczeń z praktykami zwinnymi (ponad połowa respondentów ma doświadczenie nie większe niż 2-letnie);
- równoległe stosowanie metodyk zwinnych i tradycyjnych w firmach (około 22% firm używa tylko metodyk zwinnych);
- najczęściej wykorzystywane metodyki, a mianowicie Scrum (58%) i jego modyfikacje, głównie w połączeniu z XP (17%); „czyste" XP i FDD wykorzystuje tylko 2-4% firm;
- najczęstsze przyczyny braku sukcesu projektów realizowanych metodykami zwinnymi, a mianowicie: brak doświadczeń (14%), kultura firmy (11%), zewnętrze naciski na s tosowanie metodyk kaskadowych (10%) i brak wsparcia ze strony kierownictwa (7%);
- typowe bariery we wdrażaniu metodyk zwinnych: możliwość zmiany kultury organizacyjnej firmy (51%), ogólny opór przed zmianami (40%), posiadanie pracowników o odp owiednich kwalifikacjach (40%), wspomaganie kierownictwa (34%), złożoność lub wielkość projektów (31%), współpraca klienta (29%) oraz skalowalność metodyk (25%);
- powody zastosowania metodyk zwinnych: możliwość reakcji na zmiany (82%), wzrost produktywności zespołów (80%), przyśpieszenie wejścia na rynek (78%) i wzrost jakości oprogramowania (73%);
- skrócenie czasu realizacji projektów (83%);
- najczęściej używane narzędzia, wspomagające metodyki zwinne - Excel (52%), VersionOne (36%) i MS Project (30% - *sic*!).

Wdrożenie praktyk zwinnych w firmach powinno być zatem realizowane w sposób uporządkowany jako projekt zmian organizacyjnych, obejmujący całość firmy i część jej otoczenia (np. kontakty z klientami).



Rys. 1.13. Stosowalność praktyk zwinnych
Źródło: opracowanie własne na podstawie (State, 2010)

Badania (State, 2010) dostarczyły informacji na temat najczęściej wykorzystywanych przez firmy praktyk zwinnych (rys. 1.13). Ich rezultaty są podpowiedzią w obszarze, które z praktyk należało by wdrożyć w pierwszej kolejności w firmie.

Metodyki zwinne prowadzą także do problemów międzyludzkich w firmach. Wyrażają się one utratą prywatności oraz tytułów, związanych ze stanowiskami pracy. Utrata prywatności związana jest z podejściem zespołowym, spotkaniami, pracą w jednym pomieszczeniu. Utrata tytułów (kierownik, dyrektor, główny specjalista itd.) związane jest natomiast ze spłaszczeniem struktury organizacyjnej firmy, wnoszonym przez praktyki zwinne.

Wdrożenie metodyk zwinnych miało poprawić sytuację ze skutecznością realizacji systemów informatycznych - na tyle niską, że nazwaną „chaosem". Wprowadzenie metodyk tradycyjnych i praktyk inżynierii oprogramowania zmniejszyło ten chaos. Wskaźnik projektów, które zakończyły sie sukcesem wzrósł od 1994 z 16% do 35% w roku 2006. Podobną tendencję zaobserwowano w obszarze projektów, które przerwano bez dostarczenia produktu. Ich odsetek spadł w analogicznym okresie z 56% do 19% . Niestety kolejny raport o c haosie (Chaos, 2009) wskazuje na

pogorszenie sytuacji. Odsetek udanych projektów informatycznych spadł do 32%, a tych przerwanych wzrósł do 24%. Skala wzrostu liczby nieudanych projektów w ciągu 3 lat jest zbyt duża (ponad 25%) by nie szukać przyczyn. Może być nią coraz powszechniejsze wprowadzanie do praktyki metodyk zwinnych bez odpowiedniego przygotowania.

Tabela 1.2. Czynniki wpływające na sukces projektu i ich względne znaczenie

Czynnik	**Waga**
Udział użytkownika	19
Wspomaganie ze strony kierownictwa firmy	16
Jasno zdefiniowane wymagania	15
Właściwe planowanie	11
Realistyczne oczekiwania	10
Częste punkty kontrolne	9
Kompetentni pracownicy	8
Identyfikowanie się z projektem	6
Jasna wizja i cele projektu	3
Ciężka praca, zdeterminowany zespół	3
Razem:	100

Źródło: (Chaos, 2009)

Standish Group International, poza statystykami dotyczącymi skuteczności realizacji projektów, bada także przyczyny stanu rzeczy. W tab. 1.2 przedstawione zostały czynniki, które wpływają na osiągnięcie sukcesu przez projekty informatyczne. Trzy pierwsze z nich są bardzo istotne (suma ich znaczenia wynosi 50%). Są to udział w projekcie użytkownika i kierownictwa firmy oraz jasno zdefiniowane wymagania. Dwa pierwsze czynniki są silnie akcentowane w metodykach zwinnych, a znaczna część praktyk i procesów zwinnych jest ukierunkowana na doprecyzowywanie wymagań w trakcie ich realizacji. Metodyki zwinne wychodzą bowiem z założenia, że w projektach wymagania mogą się zmieniać. Jednak brak spełnienia wymogu ustawicznej i efektywnej współpracy z użytkownikiem oraz odpowiedniego zaangażowania kierownictwa firmy (co niestety jest zjawiskiem dość częstym) może doprowadzić projekt do klęski. O dziwo, małe znaczenie dla sukcesów projektów według badań mają takie czynniki jak zdeterminowany i kompetentny zespół, identyfikacja jego członków z rezultatami prac itd. (tab. 1.2). A są to przecież jedne

z ważniejszych zasad zwinności (Shore, Warden, 2008).

Idea zwinności początkowo propagowana przez programistów zaczęła rozwijać się na obszary związane z zarządzaniem projektami. Rozwój ten doprowadził do powstania całego szeregu lekkich metodyk zarządzania projektami, coraz szerzej i coraz bardziej efektywnie wdrażanych w praktykę.

Metodyki zwinne mają cały szereg zalet, do których zaliczyć można:

- poprawa produktywności zespołów;
- pobudzenie innowacyjności, synergia myśli i pomysłów;
- znaczna poprawa zbieżności produktu z potrzebami klienta;
- wytworzenie kultury pracy zespołowej i odpowiedzialności za rezultat zespołu oraz każdego jego członka;
- wymuszenie poprawy organizacji pracy;
- wzrost zaangażowania pracowników;
- wzrost zadowolenia z pracy.

Trzeba jednak zwrócić uwagę na fakt, że „stopień" lekkości metodyki powinien być dostosowany do typu projektu. Niektóre projekty niedopuszczają bowiem do nadmiernego „odciążenia". Wymagają one dostarczenia odpowiedniej dokumentacji, zaplanowania działań, ich realizacji zgodnie z planem etc. W sukurs przychodzą w tym zakresie metodyki mieszane, łączące w sobie zalety metodyk formalnych i lekkich.

1.7. PYTANIA KONTROLNE

1. Omów problemy tradycyjnych metodyk zarządzania projektami i na tym tle ideę zwinności.
2. Przedstaw i scharakteryzuj najważniejsze wartości i zasady zwinnego podejścia do wytwarzania oprogramowania.
3. Scharakteryzuj, przedstaw cykl życia i najważniejsze elementy jednej z metodyk zwinnych (XP, Scrum, FDD).

4. W jaki sposób zarządza się zakresem prac w XP, Scrum i FDD? Jakie narzędzia się do tego stosuje?
5. Wyjaśnij ideę Młyna i wykresu wypalania. Podaj związek między nimi.
6. Przedstaw najpoważniejsze błędy i wynikające z nich problemy przy wdrażaniu metodyk zwinnych w firmie.

2

Szacowanie projektów w metodykach zwinnych

Cel

Istotą tego rozdziału jest prezentacja procesu szacowania wielkości oprogramowania w metodykach zwinnych zarządzania projektami. Poza wskazaniem na miejsce szacowania w projektach, skoncentrowano się w nim na szczegółowym przedstawieniu metryk używanych w procesie szacowania. Zaprezentowane zostały także metody szacowania, wykorzystywane w podejściu zwinnym oraz problemy zmiany ocen. Metody te są przedstawione ze wskazaniem ograniczeń oraz trudności dotyczących ich stosowania.

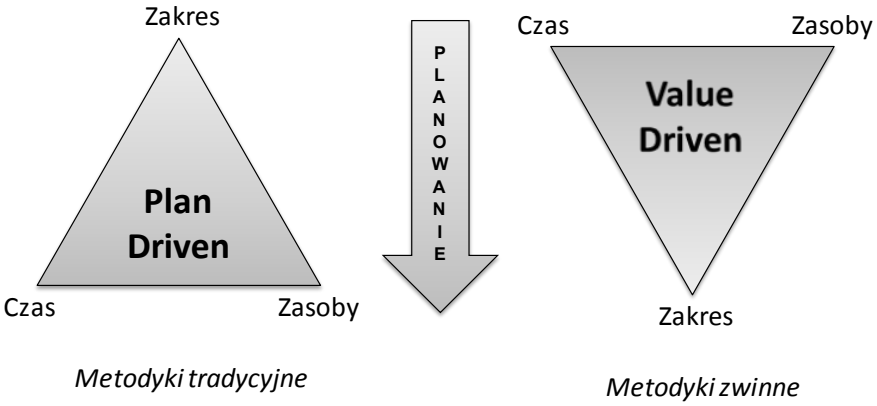
Plan

1. Miejsce szacowania w cyku życia projektu.
2. Metryki estymacji wielkości elementów projektu.
3. Techniki planowania wydań i iteracji.
4. Specyficzne metody szacowania projektów zwinnych.
5. Modyfikowanie oszacowań.

2.1. MIEJSCE SZACOWANIA W CYKLU ŻYCIA PROJEKTU

W zarządzaniu projektami informatycznymi istnieje potrzeba szacowania, czyli przybliżonego określenia (estymacji) istotnych parametrów realizacji elementów produktu, zadań czy też całego projektu. Parametry te to zwykle koszt i czas realizacji. Są to wielkości dość silnie ze sobą związane poprzez pryzmat zużycia zasobów, które przydzielane są do realizacji szacowanego elementu. W projektach programistycznych zasoby te stanowią głównie ludzie.

W metodykach tradycyjnych na podstawie ustalonego zakresu projektu planuje się czas realizacji oraz zaangażowanie zasobów. Podejście takie nazywa się zarządzaniem przy pomocy planu (ang. *Plan Driven*). Większość tradycyjnych metodyk wymaga ustalenia zakresu i stworzenia na jego podstawie mniej lub bardziej dokładnego planu. Metodyki tradycyjne mają także wbudowane mechanizmy kontroli poprawności realizacji planów i ich modyfikacji, w przypadkach kiedy okaże się to konieczne. Niemniej nadal zmiana planów następuje w oparciu o zakres działań (poprzedni lub zmieniony), na podstawie którego wyznaczany jest czas i niezbędne zasoby. Podejście zwinne odwraca trójkąt zakres-czas-zasób wprowadzając w życie zasadę szybkiego i ciągłego udostępniania wartościowego oprogramowania (rozdz. 1.2), czyli wytwarzanie sterowane wartością dla klienta (ang. *Value Driven*) - rys. 2.1.



Rys. 2.1. Różne podejścia do procesu planowania
Źródło: opracowanie własne

W podejściu zwinnym, na podstawie z góry ustalonych zasobów (np. 7. osobowy zespół w Scrum) i czasu (30. dniowy sprint), planuje się (bazując na wymaganiach - priorytetach użytkownika) zakres prac.

Przy szacowaniu projektów informatycznych należy uwzględnić miejsce szacowania w cyklu życia projektu jako wyznacznik niepewności oszacowania, jak i wyboru metody. Wraz z realizacją projektu tworzone są kolejne artefakty projektowe rozszerzające wiedzę o produkcie (specyfikacja, analizy, projekty, listy czy rejestry zadań, plany testów itd.) oraz o projekcie i zespole (dane dotyczące wydajność zespołu, rezultaty analiz ryzyka itd.). W konsekwencji dokładność szacowania zwiększa się (o ile dokonywane są kolejne oszacowania - reestymacje) w miarę zbliżania się do końca realizacji projektu. Oszacowanie projektów może obywać się (McConnell, 2006):

- we wczesnym etapie projektu – dla tradycyjnych metodyk wytwarzania oprogramowania jest to okres pomiędzy wstępną definicją projektu a określeniem wymagań, dla metodyk zwinnych – okres planowania wstępnego;
- w centralnym etapie projektu – dla tradycyjnych metodyk jest to czas od określenia wymagań projektowych – początek fazy implementacji, podczas której zbierane są dane dotyczące produktywności potrzebne do oszacowań; w metodykach zwinnych – okres od dwóch do czterech pierwszych iteracji, pozwalający na zebranie wystarczających informacji dotyczących m.in. wydajności zespołu czy poprawności początkowego oszacowania złożoności elementów (opowieści użytkownika czy też innych elementów funkcjonalności);
- w późnym etapie projektu – dla tradycyjnych metodyk etap ten zaczyna się od połowy fazy implementacji, dla metodyk zwinnych od 2 - 4 iteracji.

W projektach informatycznych konieczne jest wykonywanie ponownego oszacowania całego projektu lub jego elementów przy wystąpieniu znaczących zmian funkcjonalności, identyfikacji nowych elementów funkcjonalności w kolejnym cyklu w podejściu iteracyjnych lub pojawieniu się zmian technologicznych bądź środowiskowych mających istotny wpływ na jego realizację.

Istota wewnętrzna i potrzeby metody planowania w metodykach tradycyjnych i zwinnych są w zasadzie takie same. Polegają na wyznaczeniu potrzeb czasowo-

zasobowych projektu (lub jego części) lub też dostosowaniu zakresu do możliwości czasowo-zasobowych zespołu wykonawczego. Pierwszy przypadek można zdefiniować jako: czas+zasoby=f(zakres), a drugi: zakres=g(czas, zasoby). W istocie funkcja g jest funkcją odwrotną do f. Niezależnie od różnic, oba podejścia muszą mieć narzędzia do wyznaczania kosztów (rozumianych jako: czas+zasoby) realizacji konkretnego zakresu lub jego elementu. Wymóg ten jest niezależny od przyjętej potem metodyki bilansowania pracy i możliwości jej wykonania. Czas+zasoby w projektach informatycznych zwykle sprowadza się do pojęcia **pracochłonności** wykonania (mierzonej w np. roboczogodzinach, osobodniach czy osobomiesiącach). Pracochłonność to nic innego jak miara pracy ludzkiej (a więc jej koszt), którą trzeba wnieść by wykonać produkt.

Wszystkie metody szacowania, które wykorzystywane są w metodykach zwinnych, oparte są na trzech prostych technikach. Są nimi: analogia, dekompozycja oraz opinia eksperta (McConnell, 2006). Same oszacowania wykonywane są przez cały zespół projektowy lub jego znaczącą reprezentację (Cohn M., 2005). Dzięki czemu oszacowanie nie jest wyrazem umiejętności i wiedzy tylko jednego eksperta, ale możliwe jest wykorzystanie doświadczeń i uwag wszystkich członków zespołu pełniących w projekcie różnorodne role.

Zespoły wykonują oszacowanie z wykorzystaniem prostych i intuicyjnych technik, pozwalających na otrzymanie zadowalającej trafności oceny przy stosunkowo niewielkim wysiłku oraz kosztach. Natomiast sam dobór metody szacowania nie jest uzależniony od zastosowanego podejścia do wytwarzania oprogramowania, ani od wielkości projektu.

Metody szacowania wielkości projektu można podzielić na:

- **całościowe** - szacuje się od razu pracochłonność wykonania całego projektu bez wnikania w jego strukturę produktową i zadaniową; takie szacunki wykonuje się zwykle na bardzo początkowych etapach projektu, bardzo często w fazie przedprojektowej; do szacowania wykorzystuje sie w tym przypadku metodę przez analogię (do innych podobnych projektów) lub metodę ekspercką, realizowaną różnymi technikami (np. metodą delficką);
- **dekompozycyjne** - projekt podlega w niej podziałowi na mniejsze artefakty lub na zadania (kończące się wytworzeniem konkretnych produktów), następnie artefakty

te są rozdzielnie oceniane, by w rezultacie po zsumowaniu ocen dać oszacowanie całości.

Metoda dekompozycyjna przynosi zwykle dokładniejsze wyniki z powodu możliwości estymacji małych, jednorodnych fragmentów projektu oraz zmniejszania się błędów oszacowań w wyniku sumowań. Jest one jednak bardziej kosztowna - wymaga większej liczby działań i zaangażowania większej liczby ekspertów.

W obydwu typach oszacowań można stosować miary (metryki) **czasowe** (ang. *Time Based Estimation*), dostarczające szacunków pracochłonności, lub **względne** (ang. *Relative Estimation*). Metryki względne stosuje się zwykle w metodach parametrycznych.

Metody parametryczne wykorzystują pośrednie miary (metryki) wielkości tworzonego oprogramowania (lub jego fragmentu) i formuły (zwykle tworzone na podstawie danych empirycznych) do przeliczenia wielkości produktu (lub jego części) na istotne parametry projektu. Zwykle jest to zużycie zasobów na jego wytworzenie, czyli pracochłonność. Typowa formuła stosowana w metodach ekstrapolacyjnych ma następującą uogólnioną postać:

$$Z = f(W) \tag{2.1}$$

gdzie:

- Z - potrzebne zasoby (np. pracochłonność),
- W - estymacja wielkości w określonej metryce względnej, tj. w określonych jednostkach pomiaru,
- f - funkcja uwzględniająca warunki realizacji oprogramowania (ograniczenia, zespół i jego umiejętności, klient i jego uwarunkowania, technologia, środowisko docelowe, proces wytwórczy, czynniki zewnętrzne itd.).

Metryki względne mogą mieć różnych charakter. W konsekwencji wyróżnia się metryki ilościowe i funkcjonalne.

Metryki ilościowe operują mierzalnymi i policzalnymi jednostkami, związanymi z oprogramowaniem. W związku z czym nazywane są niekiedy metrykami i jednostkami programowymi. Jest to zwykle liczba linii kodu programu, liczba encji w logicznym modelu danych, liczba klas i ich powiązań w modelu, scenariuszy w specyfikacji wymagań czy też ilość stron w dokumentacji programu. W przypadku

serwisów internetowych mogą to być poszczególne podstrony (statyczne lub dynamiczne, mniej lub bardziej złożone) z uwzględnieniem złożoności nawigacyjnej serwisu.

Metryki funkcjonalne operują natomiast umownymi jednostkami miary, wynikającymi z funkcjonalności dostarczanej przez oprogramowanie. Oceniają one wielkość oprogramowania na podstawie jego użytecznej funkcjonalności. Zatem są one ukierunkowane na użytkownika systemu i jego potrzeby. Miarami w tych metrykach, stosowanymi często w metodykach tradycyjnych zarządzania projektami są punkty funkcyjne (ang. *Function Points*) i punkty przypadków użycia (ang. *Use Case* Points).

2.2. METRYKI ESTYMACJI WIELKOŚCI ELEMENTÓW PROJEKTU

W metodykach zwinnych do oszacowania wielkości, czyli złożoności realizacyjnej, poszczególnych elementów oprogramowania stosuje się następujące metryki:

- dni idealne (ang. *Ideal Days*) - metryka czasowa;
- punkty opowieści użytkownika (ang. *Story Points*) - metryka funkcjonalna, względna;
- rozmiar koszulkowy (ang. *T-Shirt Size*) - metryka funkcjonalna, względna.

Dni idealne (ang. *Ideal Days*) wprowadzają rozróżnienie pomiędzy, tzw. czasem idealnym (ang. *Ideal Time*) od pojęcia normalnego upływu czasu (ang. *Elapsed Time*). Czas idealny to czas potrzebny na wykonanie zadania przy założeniu, że nie są wykonywane inne zajęcia poboczne. Terminem upływu czasu określany jest czas, którego zmianę zaobserwować można na zegarze albo kalendarzu. Rozróżnienie to jest konieczne, bowiem członek zespołu projektowego nie zajmuje się jedynie wykonywaniem zadań przewidzianych w projekcie. Jego dzień roboczy, w zależności od roli w zespole, składa się również z czynności, związanych z projektem, ale nieproduktywnych, takich jak:

- spotkania i zebrania,
- prezentacje,
- wykonywanie telefonów,

- odpowiadanie na wiadomości mailowe,
- rozmowy kwalifikacyjne,
- przygotowanie i realizacja prezentacji dla klientów,
- wspieranie obecnej wersji produktu,
- poprawki błędów w obecnej wersji produktu,
- samokształcenie,
- pomoc innym członkom zespołu,
- powtarzanie już wykonanych prac (z powodu ich utracenia lub dublowania przydziału),
- przełączanie się pomiędzy zadaniami.

W ogólnym budżecie czasu pracownika (np. rocznym) należy jeszcze uwzględnić przerwy w pracy związane z jego:

- urlopami,
- chorobami,
- uczestnictwem w szkoleniach i konferencjach,
- niedyspozycjami oraz sprawami osobistymi
- czasem ustawowo wolnym od pracy.

Po uwzględnieniu tych wszystkich czynników, dysproporcja pomiędzy czasem idealnym a upływem czasu okazuje się być bardzo duża. Przykładowo, badania przeprowadzone w tym zakresie wykazują, że managerowie pracują średnio tylko 5 minut pomiędzy kolejnymi przerwaniami (Highsmith, 2004). W zależności od firmy i organizacji pracy, indywidualnych cech programisty, jego pozycji w zespole, a nawet infrastruktury technicznej straty czasu mogą osiągać 50%.

Uwzględniając różnice w wielkości efektywnego czasu pracy dla poszczególnych ról w zespole oraz chcąc uniezależnić estymacje od tych zależności, poszczególnym opowieściom użytkownika przypisywana jest wartości – ilość dni idealnych potrzebnych na ich wykonanie (tj. implementacje danego elementu funkcjonalności, przetestowanie oraz akceptacje wyników testów przez sponsora projektu).

Przy oszacowaniu w dniach idealnych pojedynczej opowieści użytkownika, zakłada się, że:

1. Dana historia użytkownika jest jedynym zadaniem, nad którym wykonywana jest praca.

2. Wszystko, co jest konieczne do wykonania zadania jest dostępne wraz z jego rozpoczęciem.
3. Podczas wykonywania zadania nie występują przerwania.
4. Organizacyjne koszty stałe wykonania zadania są pomijane.

Istotne jest również, aby przy szacowaniu danej opowieści użytkownika czy też innego elementu pracy (np. cechy) nie przypisywać liczby dni idealnych z uwzględnieniem ról w zespole, lecz traktując go jako całość. Dzięki takiemu podejściu zostanie zachowane poczucie jedności zespołu, a jest to jedna z zasadniczych wartości zespołów zwinnych. Możliwe będzie także uniknięcie konieczności śledzenia indywidualnej wydajności poszczególnych członków zespołu, a tym samym błędów estymacji, wynikających z różnic produktywności i efektywności pomiędzy osobami pełniącymi takie same role w zespole.

Każdy element projektu szacowany jest poprzez analogię do innych. Oszacowanie wielkości całego projektu jest sumą dni idealnych potrzebnych do wykonania wszystkich elementów funkcjonalności. Wykorzystywana jest zatem metoda dekompozycyjna.

Zaletą estymacji w dniach idealnych jest prostota. Wynika ona z braku konieczności uwzględniania na etapie szacowania dużej liczby czynników, skracających efektywny czas pracy poszczególnego pracownika i całego zespołu. Oczywiście czynniki te będą musiały być uwzględnione przy wyznaczaniu liczby iteracji (tj. czasu realizacji całego projektu) i przy planowaniu każdej nowej iteracji. Poza tym, jest to bardzo naturalna metoda szacowania - każdy rozumie bowiem zdanie: „zajmie mi to dwa dni”.

Do podstawowych wad dni idealnych należy użycie tych samych jednostek miary (dni) w różnym kontekście, co może doprowadzać do problemów interpretacyjnych, a także zależność estymacji od doświadczenia zespołu i wynikająca stąd konieczność częstych reestymacji. W miarę postępów prac te same elementy oprogramowania mogą wymagać mniejszej liczby dni idealnych. Istotny jest tu także negatywny czynnik psychologiczny - nie każdy bowiem jest w stanie zaakceptować fakt, że często jeden dzień pracy to zaledwie pół dnia idealnego.

Punkty opowieści użytkownika (ang. *Story Points*) są jednostkami miary do wyrażenia całkowitego rozmiaru opowieści użytkownika (ang. *User Story*), elementów

funkcjonalności (ang. *Features*) lub innej części prac, tj. tematów oraz eposów. Epos (ang. *Epic*) jest dużą historyjką użytkownika, która nie została jeszcze podzielona na mniejsze. Powodem braku takiego podziału jest najczęściej jego koszt, tzn. wcześniejsze jego oszacowanie wymaga większych nakładów pracy, bądź fakt, że epos ten nie będzie implementowany w niedługiej przyszłości. Temat (ang. *Theme*) natomiast jest zbiorem powiązanych ze sobą opowieści użytkownika traktowanych oraz planowanych jako całość. Dzięki grupowaniu opowieści użytkownika w tematy, możliwe jest zredukowanie kosztów związanych z szacowaniem. Z drugiej strony estymacja taka jest obarczona znacznie większym błędem niż estymacja dużej ilości prostych elementów czy opowieści.

Podziału elementu funkcjonalności (np. eposu lub złożonej opowieści) na mniejsze opowieści można przeprowadzić na następujące sposoby (Cohn, 2005):

- ze względu na obszary danych wspieranych przez system;
- ze względu na rodzaj wykonywanych operacji przez system;
- ze względu na priorytety mniejszych historii;
- poprzez oddzielenie aspektów funkcjonalnych i niefunkcjonalnych systemu;
- poprzez usuniecie elementów poprzecznych systemu, takich jak zabezpieczenie, uprawnienia oraz inne.

W trakcie szacowania do każdego elementu funkcjonalności przypisywane są punkty. Przypisana wartość nie jest istotna, ważna jest jej relatywna wielkość. Punkty są bowiem przypisywane do danego elementu przez analogie do innych. Nie istnieje żaden wzór opisująca zbiór wartości przypisywanych do opowieści użytkownika. Zbiór ten powinien odzwierciedlać skalę wartości wyrażających wysiłek stworzenia danej funkcjonalności (w tym przygotowania i przeprowadzenia testów), będący wypadkową takich wartości jak: złożoność, ryzyko, uwarunkowania realizacyjne oraz inne czynniki związane z danym elementem funkcjonalności.

Istnieją dwa sposoby rozpoczęcia szacowania, tj. wyboru punktu odniesienia. Pierwszy polega na wybraniu najmniejszego elementu (np. opowieści użytkownika) oraz przypisaniu mu wartości równej 1. Drugi, na wybraniu opowieści o średniej złożoności całkowitej i nadaniu jej wartości ze środka używanej skali. Kolejne elementy szacowane są przez porównanie z uprzednio wybranymi elementami.

W projektach realizowanych metodykami zwinnymi możliwe jest rozpoczęcie

iteracji z niekompletną specyfikacją, której szczegóły zostaną określone dopiero podczas danej iteracji. Niemniej w takim wypadku konieczne jest oszacowanie niedoprecyzowanych elementów funkcjonalności, nawet jeśli rezultat będzie znacznie mniej dokładny.

Najlepszą skalą do estymacji jest skala oparta o jeden rząd wielkości. Najczęściej są stosowane dwie następujące skale:

- 1, 2, 3, 5 i 8, czyli ciąg Fibonacciego (bez pierwszego elementu);
- 1, 2, 4 i 8, czyli ciąg złożony podwójnej wartości elementu go poprzedzającego.

Obydwie skale najlepiej odwzorowują niepewność związaną z szacowaniem większych jednostek pracy. Małe jednostki mogą być szacowane z dużą dokładnością. W przedziale początkowym skali powinno znaleźć się więcej punktów (liczb). Duże jednostki, z racji ich złożoności, braku wiedzy o strukturze i zawartości, nie mogą być szacowane tak dokładnie. W związku z tym skala narasta nieliniowo. Podczas szacowania możliwe jest również użycie wartości 0. Jest to stosowane w sytuacji, gdy zespół uzna dany element funkcjonalności za wymagający znikomego nakładu pracy. Dodanie zera do skali można uznać za wyjątkowo przydatne, jeśli estymacja ma zawierać się w skali zdziesięciokrotnionej (np. 10, 20,...) albo, jeśli oszacowana wielkości zadania zbliżona jest bardziej do 0. niż do 1. W takich przypadkach zespół nie zwiększa sztucznie ocen. Należy jednak pamiętać, że zadania oszacowane na 0 również wymagają pracy. Jest to szczególnie ważne, gdy takich zadań jest kilka czy kilkanaście. Dobrym rozwiązaniem takiego problemu na koniec procesu szacowania jest zgrupowanie kliku zadań oszacowanych na 0 i nadanie im wartości 1 (Cohn, 2005).

Niektóre zespoły preferują pracę na większych liczbach. Mogą one korzystać ze skal odpowiednio pomnożonych przez 10. W tym wypadku należy jednak wystrzegać się złudnej precyzji, jaką może dostarczać taka skala. Szacowanie historyjek na 35 czy też 37 nie oddaje różnicy pracochłonności pomiędzy zadaniami, a raczej zaciera podobieństwa i różnice, które należałoby wskazać. Dlatego też bardziej odpowiednie są liczby jak: 10, 20, 40, 80, 100 czy też 10, 20, 30, 50, 80, 100.

W przypadku szacowania eposów albo tematów należy dodać kolejne elementy do skali według odpowiedniego wzorca lub odpowiednio dobranych wartości. Sugeruje się, dodanie elementów takich jak 13, 21, 34 oraz 89 (kolejne liczby Fibonacciego).

Przyjęcie dodatkowych wartości rozciąga skalę i dlatego często stosowana jest ona po modyfikacjach zawężających - np. do wartości 1, 2, 3, 5, 13, 40, 100. Ostatnia wartość wskazuje na element (zwykle epos) o zbyt dużej wielkości, by można było go właściwie szacować i kontrolować. Element taki powinien być poddany analizie, podziałowi i oszacowaniu fragment po fragmencie. Fragmenty powinny jednak stanowić zamknięte, skończone całości. Zawężone skale mają zatem bardzo praktyczny sens - dość dokładnie można oszacować względne wielkości małych elementów a dużych i tak nie warto, bo będą dekomponowane. Odwzorowują one sytuacje: „to jest 3 razy większe od tamtego, ale ten element jest bardzo duży" (szacunki: 3, 1, 100).

Metryką analogiczną do punktów opowieści użytkownika jest **rozmiar koszulkowy** (ang. *T-Shirt Size*). W niej używa się popularnych oznaczeń, stosowanych do określana w handlu wielkości koszulek (ang. *T-Shirts*). Są to: S/M/L/XL/XXL. Metryka ta jest bardziej zrozumiała (jest doskonałą metaforą rzeczywistości), ale też może być uważana za śmieszną i nienaukową. Tym niemniej jest coraz częściej używana. Proponowane są także (Foy, 2011) pewne standardowe przeliczanie rozmiaru koszulkowego na liczbę jednodniowych cykli, niezbędnych do zrealizowania wymagania - tab. 2.1.

Tabela 2.1. Czas realizacji opowieści użytkownika o różnych wielkościach

	Liczba cykli (dni)
Mała (S)	4
Średnia (M)	15
Duża (L)	27

Źródło: (Foy, 2011)

Metryki punktów opowieści użytkownika i rozmiar koszulkowy mają jedną bardzo poważna wadę. Jest nią brak możliwości porównywania wielkości oprogramowania szacowanego przez różne zespoły, a co za tym idzie brak możliwości porównywania produktywności zespołów. Każdy zespół bowiem samodzielnie ustala początkową wielkość szacowanego elementu, względem którego szacowane są pozostałe. Taki brak standaryzacji uniemożliwia również uogólnianie doświadczeń i wymianę wiedzy na temat estymacji projektów zwinnych. Dane pojawiające się w publikacjach są

danymi jednostkowymi i przykładowymi (Foy, 2011) i (Dajda, 2008).

Podstawową zaletą metryk punktów opowieści użytkownika jest ich niezależność od produktywności zespołu. Po jej zmianie szacunek ilości pracy do wykonania nie ulega zmianie. Ogranicza to liczbę ponownych estymacji projektu i jego części. Poza tym zespoły zwykle łatwiej uzgadniają szacunki mierzone w punktach niźli konkretnych, czy też idealnych, dniach. Wadą są problemy z wyjaśnieniem przyjętej skali - chociaż rozmiar koszulkowy likwiduje częściowo ten problem

Szacowanie z użyciem dni idealnych i punktów opowieści użytkownika jest zazwyczaj przeprowadzane tą samą techniką — przez analogię. Najbardziej istotne różnice pomiędzy tymi metodami przedstawione zostały w tab. 2.2.

Tabela 2.2. Porównanie jednostek szacowania w metodykach zwinnych

	Ideal Days - dni idealne	**Story Points - punkty opowieści**
Zalety	Łatwiejsze do nauki dla zespołów rozpoczynających pracę metodykami zwinnymi. Bardziej zrozumiałe dla osób spoza zespołu, takich jak sponsor, klienci i decydenci.	Niezależne i abstrakcyjne jednostki. Oszacowanie nie zmienia się wraz ze zmianą produktywności zespołu. Oszacowanie wspiera zachowanie wielofunkcjonalne.
Wady	Oszacowanie staje się nieaktualne wraz ze wzrostem wydajności zespołu. Jednostka, zależy od zdolności i preferencji danego członka zespołu czy też całej grupy.	Trudniejsze do zrozumienia dla decydentów.

Źródło: (Opracowanie własne)

2.3. TECHNIKI PLANOWANIA WYDAŃ I ITERACJI

W metodykach zwinnych planowanie wydań iteracji realizowane jest z wykorzystaniem zasady dostarczenia klientowi jak najbardziej wartościowego dla niego oprogramowania. Jest to planowanie sterowane wartością dla klienta (rys. 2.1).

W zależności od metodyki zwinnej na etapie planowania wydania rozwiązywane jest jedno z zadań:

- wyznaczenie czasu trwania wydania na podstawie ustalonej listy elementów funkcjonalności (opowieści użytkownika, cechy);
- wybranie, które elementy funkcjonalności uda się zrealizować w zadany czas (np. sprintu w metodyce Scrum).

Czas trwania wydania obliczany jest jako iloraz wartości oszacowania wszystkich elementów funkcjonalności przewidzianej do realizacji w danym wydaniu oraz średniej, szacowanej wydajności zespołu:

$$T_w = \frac{\sum_{i \in R} w_i}{\overline{V}_w}, \quad w = 1, 2, ... \tag{2.2}$$

gdzie:

T_w - czas realizacji wydania o numerze w;

w_i - estymacja wielkości *i*-tego elementu, zaplanowanego do realizacji w danym wydaniu;

R - zbiór elementów, zaplanowanych do realizacji w danym wydaniu;

$\overline{V}_w$ - średnia wydajność całego zespołu dla wydania o numerze w, oszacowana w jednostkach metryki pomiaru wielkości elementów (np. punkty opowieści użytkownika) na jednostkę czasu (np. dzień, iterację czy sprint).

Średnia wydajność zespołu, zwana w metodykach zwinnych **szybkością zespołu** (ang. *Team Velocity*), mierzona jest liczbą punktów opowieści użytkownika (lub rozmiaru koszulkowego) na dzień (lub iterację czy sprint) albo jako stosunek liczby idealnych dni do dni pracy. Dotyczy ona całego zespołu w analizowanym okresie. Jest więc sumą indywidualnych wydajności członków zespołu w zadanym okresie.

Sposób postępowania przy wyznaczaniu czasu realizacji wydania przedstawia schemat na rys. 2.2.

Rys. 2.2. Schemat szacowania projektu w metodykach zwinnych
Źródło: (Cohn, 2005)

Wzór (2.2) może być również użyty do wyznaczania:

- **estymowanego czasu trwania całego projektu** - w tym przypadku brane są pod uwagę wszystkie zaplanowane w projekcie elementy, a nie tylko te z jednego wydania (jeśli w projekcie przewidziano wiele wydań); uwzględniana jest też stała średnia wydajność zespołu; szacowanie to w metodykach zwinnych nosi bardzo przybliżony charakter, ze względu na istotę podejścia (uściślenie zakresu projektu jest odkładane w czasie) oraz niedokładne szacowanie znanych, ale dużych elementów;
- **szacowanej liczby wydań** w projekcie (lub sprintów w Scrum) - w tym przypadku średnia wydajność zespołu dotyczy jednego wydania (sprintu), a suma oszacowań wielkości elementów - całego projektu.

Średnia wydajność zespołu ma zatem fundamentalne znaczenie zarówno podczas planowania wydania produktu, jak i poszczególnych iteracji. Wydajność zespołu można oszacować przy pomocy jednej z następujących metod (Cohn, 2005):

- wykorzystanie z danych historycznych,
- przeprowadzenie iteracji szacującej,

- dekompozycja i synteza.

Szacowanie wydajności na podstawie **danych historycznych** zespołu, czyli wykorzystanie posiadanych danych ze zrealizowanego projektu w nowym, jest bardzo dobrą metodą. Jest ona jednak bardzo rzadko używana, ponieważ aby można było wykorzystać wartości historyczne, różnice pomiędzy starym a nowym projektem muszą być niewielkie. Szczególne znaczenie ma w tej metodzie zgodność technologiczna, dziedziny projektu, składu zespołu projektowego oraz właściciela produktu (czy jest nim ta sama osoba). Również istotna jest zgodność w obydwu projektach używanych narzędzi programistycznych oraz środowiska pracy, a także to czy estymacja była wykonywana przez ten sam zespół ludzi. Jeśli któryś z wymienionych czynników jest różny, należy poważnie zastanowić się nad skutecznością tej metody, albo do oszacowanej wielkości dodać przedział (traktowany jako bufor) niepewności. Szerokość tego przedziału wyznacza poziom błędu rezultatu. W początkowej fazie projektu, gdy niepewność oszacowań jest największa, należy użyć bufora niepewności w wysokości od 60% do 160% otrzymanej wartości, tzn. przyjąć w planowaniu wartość estymowaną ±60%. Jeśli nowy projekt jest zbliżony do już wcześniej zrealizowanego, możliwe jest korzystanie z estymacji na podstawie wartości historycznej jako z oszacowania stabilnego, tj. bardzo dokładnego na początku, ale i dalej w projekcie.

W sytuacji, gdy nie jest możliwe użycie wartości historycznej albo zespół informacji tych nie posiada, idealnym rozwiązaniem jest realizacja **iteracji szacującej** przez zespół, przy czym 2, 3 iteracje pozwalają uzyskać bardziej precyzyjne dane. Wydajność estymuje się na podstawie realnej, zaobserwowanej wartości wydajności zespołu. Częstą praktyką w projektach prowadzonych metodykami zwinnymi jest wstrzymanie się z oszacowaniem projektu do czasu przeprowadzenia przynajmniej jednej iteracji, co jest zgodne z tą metodą. Należy zauważyć, że w projektach realizowanych metodykami tradycyjnymi zadania analizowania wymagań, tworzenia listy zadań oraz planu projektu zajmują co najmniej tyle samo czasu co kilka iteracji w projektach zwinnych. Jeśli do oceny wydajności została wykorzystana tylko pojedyncza iteracja, to należy uwzględniać w dalszych szacunkach przedział niepewności ±60%. Wyniki analizy szybkości zespołu w kolejnych iteracjach umożliwiają zwiększenie dokładności szacunków. Wyznacza się bowiem średnią

wydajność w dwóch, trzech i więcej iteracjach, zawężając przedział niepewności wg. danych z tab. 2.3. Jak wynika z tab. 2.3 przy dużej liczbie iteracji i stosowaniu uśrednienia można osiągnąć 10% dokładność estymacji średniej wydajności zespołu.

Tab. 2.3. Tabela wartości mnożników przedziału niepewności

Liczba iteracji	Mnożnik ograniczenia dolnego	Mnożnik ograniczenia górnego
1	0,6	1,60
2	0,8	1,25
3	0,85	1,15
4 lub więcej	0,9	1,10

Źródło: (Cohn, 2005)

Cześć zespołów zwinnych, podczas projektów z przynajmniej trzema iteracjami badającymi wydajność zespołu, jako oszacowanie używa przedziału $\langle a,b \rangle$, gdzie a oznacza najmniejsza wartość wydajności osiągniętej podczas wszystkich iteracji, a b - odpowiednio największa wartość.

Metoda dekompozycji i syntezy używana jest w przypadku braku wartości historycznych i kiedy nie jest możliwe przeprowadzenie iteracji w celu uzyskania wydajności zespołu. Powodów, dlaczego nie można użyć metody oszacowania na podstawie zrealizowanej iteracji, jest kilka. Najczęściej dzieje się tak w przypadku estymacji tych projektów, które zostaną rozpoczęte dopiero za kilka miesięcy albo projektów, których kontrakt nie został jeszcze podpisany przez klienta. W metodzie tej należy podzielić (zdekomponować) opowieści użytkownika na zadania elementarne (dość proste, a więc łatwo estymowane) oraz oszacować czas ich trwania w roboczogodzinach. Są to działania analogiczne do tych realizowanych podczas planowania sprintu w Scrum, tj. tworzenia rejestru zadań sprintu (rys. 1.8). Następnie należy oszacować liczbę godzin, jaką każdy członek zespołu projektowego będzie dostępny każdego dnia iteracji. Jeśli zespół nie został jeszcze stworzony, należy dyrektywnie określić liczebność zespołu na podstawie potrzeb projektu, metodyki jego realizacji lub dostępności. Poprzez sumowanie należy określić całkowitą liczbę godzin

pracy zespołu dla pojedynczej iteracji. Należy przy tym pamiętać, że uwzględnia się przy tym godziny efektywnej pracy, a nie „bycia w firmie". Podobnie jak ma to miejsce przy szacowaniu dni idealnych. Następnie dowolnie lub w sposób losowy należy wybierać opowieści użytkownika, aż do wybrania wystarczającej ilości zadań, które można zrealizować podczas pojedynczej iteracji. Suma wyznacza wydajność w trakcie iteracji. Można ją łatwo przeliczyć na wydajność zespołu na jeden dzień roboczy (dzieląc przez liczbę dni roboczych w iteracji). W ten sposób uśrednia się możliwości wykonawcze członków zespołu, uzyskując średnią wydajność (szybkość) zespołu. Stosując metodę dekompozycji i syntezy należy zwrócić uwagę na wybór opowieści użytkownika. Najlepszy jest całkowicie losowy. Jeśli jednak są one wybierane w sposób dowolny (tj. dyrektywny przez kogoś), to wskazane jest zachowanie różnorodności tych elementów. Zbiór wybranych opowieści powinien zawierać zadania realizowane przez różnych członków zespołu i różnorodne co do zawartości merytorycznej, technologicznej itd.

W metodyce Scrum rozdziela się planowanie produktu od planowania sprintu. **Planowanie produktu** polega na stworzeniu (osoba odpowiedzialna: Właściciel produktu) rejestru produktu, czyli uporządkowanej (wg. wartości dla klienta) opowieści użytkownika (lub też eposów). Każdy element na liście jest szacowany w punktach opowieści użytkownika (lub dniach idealnych). Suma punktów wyznacza wielkość całego projektu i jest używana do szacowanie liczby sprintów zgodnie ze wzorem (2.2) i procedurą z rys. 2.2.

Planowanie sprintu w metodyce Scrum odbywa się w warunkach ustalonej długości sprintu (zwykle 1 miesiąc) i przy założeniu, że sprint ma się zakończyć produktem gotowym do eksploatacji. Odbywa się ono przy pomocy gry planistycznej (opisanej w rozdz. 1.3) lub innej techniki. Polega na zbudowaniu rejestru sprintu, możliwego do wykonania w danym sprincie. Możliwości ocenia się na podstawie wydajności zespołu w sprincie (mierzonej w punktach opowieści użytkownika dla całego sprintu lud dniach idealnych możliwych do wykonania w sprincie). Po stworzeniu rejestru sprintu, każda opowieść użytkownika jest dzielona na zadania, których czas realizacji jest szacowany w roboczogodzinach (idealnych). Równolegle szacowany jest budżet czasu każdego pracownika w sprincie (również w idealnych roboczogodzinach):

$$v_j^w = \frac{n_j * l^w}{o_j} \tag{2.3}$$

gdzie:

v_j^w - szacowana wydajność (szybkość) j-tego pracownika w sprincie o numerze w w idealnych roboczogodzinach;

n_j - liczba godzin dziennie pracy j-tego pracownika;

l^w - czas trwania sprintu w dniach roboczych;

o_j - współczynnik obciążenia j-tego pracownika, liczony jako stosunek roboczogodzin do idealnych roboczogodzin (zwykle w przedziale 1,5-2).

W trakcie planowania sprintu pracownik podejmuje się wykonania zadań do wysokości jego budżetu czasu w danym sprincie. Śledzenie wykonania zadań dostarcza danych do sprawozdawczości oraz do zmiany estymacji zarówno opowieści użytkownika jak i zadań.

2.4. SPECYFICZNE METODY SZACOWANIA PROJEKTÓW ZWINNYCH

W projektach realizowanych metodykami zwinnymi stosowane są również specyficzne metody szacowania, takie jak Wideband Delphi i Planning Poker.

Metoda Wideband Delphi (WBD) jest rozszerzeniem powszechnie znanej delfickiej metody szacowania (McConnell, 2006). Wideband Delphi jest metodą ekspercką rozszerzoną o dekompozycje, w której oszacowanie jest iteracyjnie wykonywane przez cały zespół projektowy. Szczególnym atutem tej metody jest to, że oszacowanie dokonywane jest przez zespół, który będzie realizował ten projekt, bazując na osobistym doświadczeniu jego członków. Doświadczenie to dotyczy zarówno dziedziny projektu, jak i obszaru technicznego. Oszacowanie jest wykonywane w kilku cyklach, podczas których wszyscy członkowie zespołu (eksperci) mogą zgłaszać uwagi oraz spostrzeżenia mające wpływ na wielkość oszacowania. Metoda ta pozwala na opracowanie adekwatnej oraz precyzyjnej estymacji projektu

dla danego zespołu (Thompson, 2011).

Oszacowanie metodą WBD składa się z całego szeregu działań (rys. 2.3), których dwa są najważniejsze: spotkanie inicjującego oraz sesja estymacji.

Rys. 2.3. Działania w metodzie Wideband Delphi
Źródło: opracowanie własne na podstawie (Wiegers, 2007)

Spotkanie inicjujące (ang. *Kick-off Meeting*) prowadzi moderator wybrany przez kierownika projektu. W spotkaniu bierze udział zespół oszacowujący. Składa się on z 3. do 7. członków zespołu projektowego reprezentujących różne role w zespole. O ile kierownik projektu uzna za stosowne, w skład zespołu mogą wchodzić także inne osoby zainteresowane powodzeniem projektu. Moderator przedstawia zespołowi oszacowującemu informacje na temat celu projektu, wizji produktu, zasobów projektowych oraz jednostek (metryk), w których projekt będzie szacowany. Moderator dostarcza również uczestnikom spotkania niezbędną dokumentację oraz listę elementów funkcjonalności, która będzie szacowana.

Pomiędzy spotkaniem inicjującym a sesją estymacji powinien być zapewniony odstęp czasowy, który pozwoli ekspertom samodzielnie zapoznać się z wymaganiami projektu (rys. 2.3).

Sesja estymacji rozpoczyna się dyskusją członków zespołu na temat założeń oraz ewentualnych problemów z oszacowaniem. Eksperci dzielą się uwagami i spostrzeżeniami. Następnie każdy członek zespołu indywidualnie opracowuje oceny elementów funkcjonalności zapisując uzasadnienie oszacowania oraz uwagi. Ważne jest, aby wszystkie estymacje wykonane były anonimowo. Przykładowa karta do ocen (formularz) została przedstawiona na rysunku 2.4.

Ekspert: Magda Data: 2/08/2011 Nr formularza: 1/1

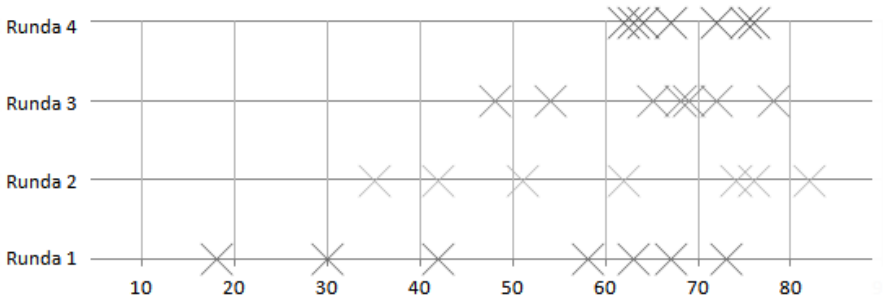
Zdefiniowany cel: Stworzenie testów on-line Jednostki: dni

Lp.	Element funkcjonalności	Oszac.	Różnica 1	Różnica 2	Różnica 3	Różnica 4	Wynik	Założenia
1	Nauczyciel może utworzyć nowy test	3	+2	+1				
2	Nauczyciel może dodać pytanie do testu	10	+5	-2	-1			4 typy pytań
3	Student może rozwiązać test	3						
4	Student może sprawdzić wyniki	1	+1	+2	+1			
5	Nauczyciel może zakończyć test	1						
6	Nauczyciel może sprawdzić wyniki studentów	5	+2	+2	-1			też graficznie
	Różnica		+10	+3	-1			
	Razem	23	33	36	35			

Rys. 2.4. Przykładowy formularz estymacji w metodzie WBD

Źródło: opracowanie własne

Wypełnione karty oszacowań są zbierane przez moderatora, który następnie przedstawia wyniki pierwszej tury estymacji w formie raportu. Elementem raportu jest wykres, przedstawiający zestawienie ocen poszczególnych ekspertów - rys. 2.5. Uwidaczniane są w ten sposób rozbieżności ocen, ale bez przypisania oceny do poszczególnej osoby.



Rys. 2.5. Wykres rezultatów estymacji w metodzie WBD

Źródło: opracowanie własne na podstawie (Wiegers, 2007)

Wykres rezultatów estymacji umożliwia każdemu ekspertowi porównanie własnego stanowisko z innymi i jego krytyczna ocenę. Każdy członek zespołu ma możliwość zapoznania się także z założeniami i uwagami innych ekspertów, ale bez

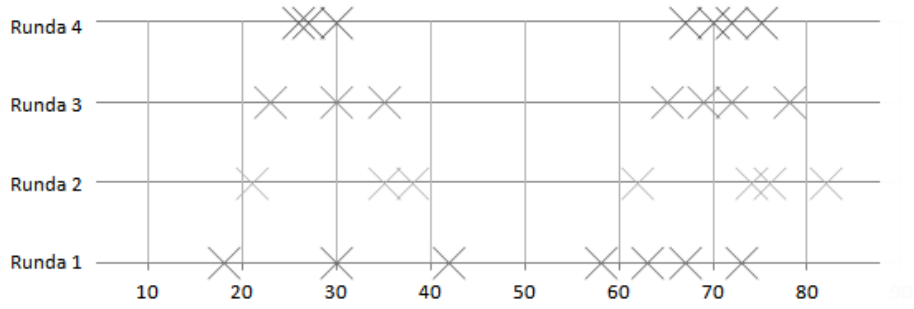
możliwości przypisania ich do konkretnej osoby. Przed kolejną turą estymacji uczestnicy spotykają się ponownie by przedyskutować nowe spostrzeżenia i refleksje. Podczas dyskusji omawiane są elementy funkcjonalności, a nie wielkości poszczególnych oszacowań. Po dyskusji eksperci modyfikują swoje zdanie i zapisują na formularzu (rys. 2.4) zmiany (na plus i minus). Po czym moderator ponownie opracowuje raport z rundy estymacji i umieszcza je wykresie (rys. 2.5).

Procedura indywidualnego szacowania, zbierania i opracowywania wyników przed moderatora oraz omawiania różnic jest powtarzana wielokrotnie. Kryteriami jej zakończenia są (Wiegers, 2007):

- wykonano 4 rundy estymacyjne - zwykle po tylu etapach, zdania członków zespołu sie już nie zmieniają;
- oceny zeszły się tak blisko, że różnice między nimi są mniejsze od uzgodnionych na etapie planowania;
- wyczerpano czas sesji estymującej - wskazuje to bowiem na „jałowe", nic nie zmieniające dyskusje;
- wszyscy eksperci nie chcą już zmieniać swoich ocen - dalsze rundy nic nie wnoszą.

Uzyskane w ten sposób szacunki są potem uogólniane, tj. uśredniane dla pojedynczego elementu funkcjonalności oraz sumowane dla całego projektu, a potem prezentowane zespołowi w celu ostatecznej oceny rezultatów (rys. 2.3).

Metoda ta jest skuteczna, ponieważ uśrednia błędy poszczególnych oceniających, dając jedno oszacowanie wypracowane przez grupę wykonującą projekt. Jednak jej wynik nie może być użyty jako oszacowanie dla innego zespołu, a zatem w jakikolwiek sposób uogólniany czy standaryzowany. Wadą tej metody może być również trudność w otrzymaniu bezstronności oszacowań, co może mieć miejsce, jeśli moderatorem nie jest odpowiednio dobrana osoba. Niekiedy metoda Wideband Delphi nie doprowadza do uzgodnienia wyniku - rys. 2.6. W takim przypadku należy podjąć działania poszukujące przyczyny rozbieżności.



Rys. 2.6. Polaryzacja estymacji ekspertów w rozbieżnych punktach w metodzie WBD
Źródło: opracowanie własne

Planning Poker jest grą planistyczną wykorzystywaną do szacowania wielkości projektu. Metoda ta jest kombinacją następujących technik: opinii eksperta, dekompozycji i analogii. Dostarcza ona dobre oszacowania w krótkim czasie (Haugen, 2011).

Planning Poker jest prowadzony przez moderatora, którym jest zazwyczaj właściciel produktu lub analityk. Moderator nie bierze udziału w szacowaniu, jego zadaniem jest czytanie opisów ocenianych elementów funkcjonalności, odpowiadanie na pytania oraz koordynowanie pracy zespołu podczas szacowania. Pozostali uczestnicy to członkowie zespołu projektowego. Jeśli zespół projektowy jest większy niż 10 osób, choć jest to rzadka praktyka w zespołach zwinnych, powinien on zostać podzielony na mniejsze zespoły pracujące niezależnie.

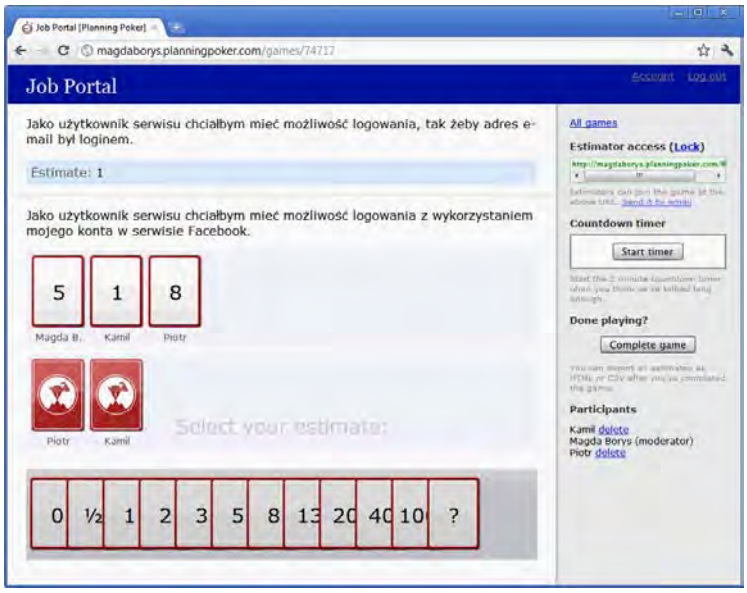
Zespoły wykonują oszacowania elementów funkcjonalności używając do tego wcześniej przygotowanych kart. Karty zawierają liczby w wybranej przez zespół skali do szacowania oraz jednostkach, liczby te powinny być odpowiednio duże, aby były widoczne dla każdego z uczestników. Każdy uczestnik posiada własny zestaw kart.

Po przeczytaniu przez moderatora opisu opowieści użytkownika oraz ewentualnym odpowiedzeniu na pytania, uczestnicy wybierają kartę z liczbą wskazującą ich oszacowanie. Podnoszą ją dopiero, gdy wszyscy uczestnicy dokonają wyboru. Zazwyczaj oszacowania różnią się, dlatego też osoby, które wybrały najniższe i najwyższe oszacowanie uzasadniają pozostałym swój wybór. Zespół może także przedyskutować poszczególne estymacje. Jeśli moderator uzna za stosowne

oraz pomocne, może wykonywać notatki z przebiegu dyskusji. Po dyskusji następuje ponowne oszacowanie - uczestnicy ponownie wybierają karty. Grupa może zadecydować o nadaniu opowieści danego oszacowania, jeśli wyniki są zbieżne. Planning Poker jest wykorzystywany nie tylko do szacowania wielkości projektu przed jego rozpoczęciem. Tą samą techniką są często estymowane nowe zadania zidentyfikowane podczas trwania iteracji.

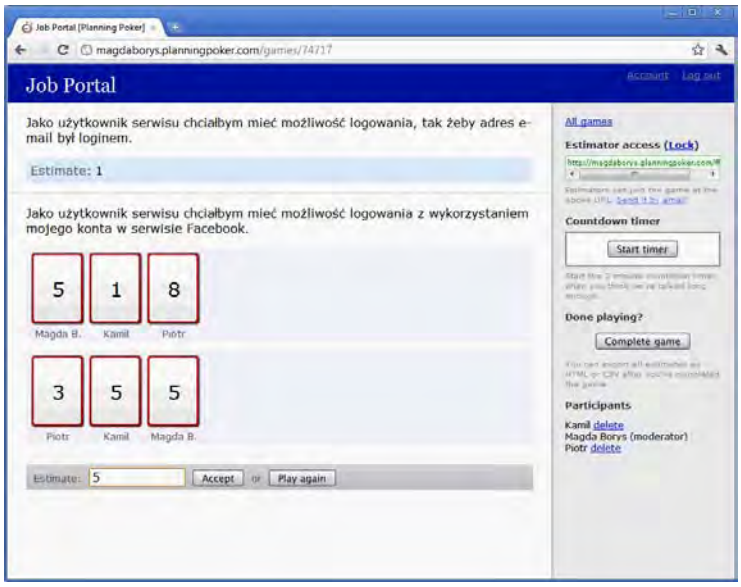
Oszacowanie, wykonane przez zespół realizujący dany projekt, a więc grupę ekspertów z poszczególnych dziedzin, jest zawsze lepsze niż oszacowanie pojedynczego analityka. Tak też jest w przypadku *Planning Poker*. Również dzięki takiemu spotkaniu możliwe jest zidentyfikowanie brakujących informacji oraz poszerzenie wiedzy o produkcie.

W przypadku, gdy członkowie zespołu znajdują się w rożnych lokalizacjach, oszacowanie z wykorzystaniem metody Planning Poker można również przeprowadzić korzystając z dostępnych aplikacji internetowych. Przykładowe oszacowanie projektu z wykorzystaniem aplikacji stworzonej przez Mountain Goat Software zostało zaprezentowane na rys. 2.7 i 2.8 (Mountain Goat Software, 2011).



Rys. 2.7. Planning Poker - szacowanie opowieści użytkownika

Źródło: opracowanie własne



Rys. 2.8. Planning Poker – wybór wartości oszacowania dla opowieści

Źródło: opracowanie własne

2.5. MODYFIKOWANIE OSZACOWAŃ

W trakcie realizacji projektu może okazać się, że pierwotne oszacowanie jest nieadekwatne do trudności oraz ilości wykonanej pracy. Należy wówczas dokonać ponownego oszacowania. Reestymowane powinny być nie tylko elementy wykonane, ale również wszystkie elementy analogiczne, tak aby posiadane wielkości oszacowań pracy wykonanej w projekcie, jak i wielkości pracy pozostającej do ukończenia systemu, były poprawne. Ponownej estymacji dokonuje się podczas spotkania inicjującego kolejną iterację lub na spotkaniu podsumowującym wydanie, czy też projekt.

Czas trwania projektu otrzymywany jest jako wynik dzielenia wielkości projektu przez wydajność zespołu realizującego projekt (wzór 2.2). Wartość ta zależna jest więc od trafności estymacji wielkości projektu. Jednocześnie, w przypadku zmian wielkości przedsięwzięcia, nie jest konieczne wykonywanie skomplikowanych

oszacowań czasu, a jedynie dokonanie powtórnych obliczeń na podstawie przypuszczalnej wielkości projektu i wydajności zespołu.

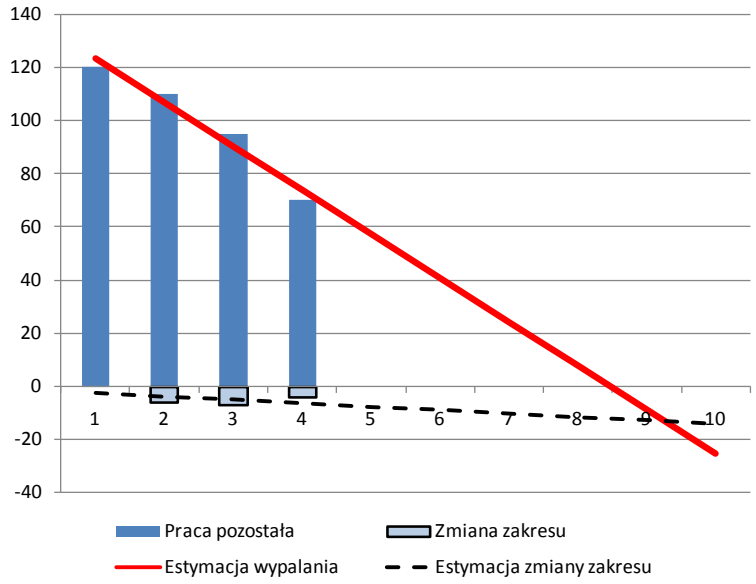
Jednym ze sposobów ominięcia problemu reestymacji jest odpowiednia zmiana wydajność zespołu w przypadku, gdy jednostka miary o wartości 1. okazuje się większa bądź mniejsza (wymaga więcej lub mniej pracy) niż zakładano. Zmieniając odpowiednio wartość tego wskaźnika możliwe jest unikniecie kosztów powtórnego oszacowania. Wynikające z tych poprawek zmiany wielkości są pomocne do obliczenia nowego, poprawnego czas trwania.

W przypadku zespołu, który stwierdził, że bezwzględna oszacowana wartość danej opowieści użytkownika jest nieprawidłowa, niezbędna jest **reestymacja częściowa**. Zespół ten powinien zmieniając wartość opowieści, zmienić także wartość oszacowania innych elementów funkcjonalności z nią związanych. Na przykład, jeśli jedna z opowieści użytkownika okazuje się przeszacowana i zespół stwierdził, że należy zmniejszyć dwukrotnie jej wartość, dwukrotnie także powinna zostać zmniejszona wartość wszystkich podobnych elementów.

Oszacowanie powtórne jest także koniecznością, gdy zmieni się względny rozmiar danej opowieści. Na przykład, gdy dana opowieść użytkownika zawiera element trudniejszy do implementacji niż poprzednio szacowano, a pojawia się on również w innych opowieściach jako ich cześć. Kolejnym powodem może być przymus podziału opowieści na mniejsze fragmenty, tak aby mogły być one wykonane w pojedynczej iteracji — eposy oraz tematy są najczęściej zbyt duże na pojedynczą iterację lub do zatwierdzonych elementów funkcjonalności możliwe jest dodanie tylko fragmentu innej opowieści.

Analogicznie, reestymacja może być wykonana, gdy zespołowi nie udało się dokończyć w pojedynczej iteracji elementu funkcjonalności (opowieści użytkownika), a chce on wartość wykonanego fragmentu doliczyć do wydajności osiągniętej podczas tej iteracji.

Rejestracja wykonanej pracy oraz modyfikacja jej oszacowań może służyć do estymacji liczby sprintów czy też iteracji do zakończenia wydania. Służy do tego **wykres wypalania wydania**, który pokazuje sprint po sprincie, ile pracy w projekcie (np. w punktach opowieści użytkownika) jeszcze pozostało do wykonania rejestru produktu. Na rys. 2.9 przedstawiono taki wykres.



Rys. 2.9. Wykres wypalania wydania i zmiany zakresu
Źródło: opracowanie własne

Słupki na wykresie spalania wskazują na pracę, którą jeszcze należy wykonać, a ujemne wartości to dodatkowa praca, która pojawiła się w danym sprincie. Linie wskazują na prognozę „wypalania" i prognozę zmiany zakresu. Przecięcie linii pozwala oszacować liczbę sprintów (w tym przypadku: 9 lub 10).

2.6. PYTANIA KONTROLNE

1. Kiedy powinno być wykonywane oszacowanie elementów funkcjonalności?
2. W jaki sposób jest szacowany czas trwania projektu w metodykach zwinnych?
3. Kiedy konieczne jest wykonanie dekompozycji elementu funkcjonalności podczas szacowania? Podaj przykłady w jaki sposób może być ona wykonana.
4. Podaj przykłady jednostek oszacowania w metodykach zwinnych.
5. Jakie są metody obliczania wydajności zespołu w metodykach zwinnych?

3

Zarządzanie jakością w procesie wytwarzania oprogramowania

Cel

Celem rozdziału jest przedstawienie mechanizmów i technik wspomagających osiągnięcie wysokiej jakości oprogramowania. Zaprezentowane zostały wymiary jakości i podstawowe zasady ułatwiające tworzenie wydajnego kodu. Rozdział zawiera opis metodyk zarządzania jakością, w tym przeprowadzania różnego rodzaju testów. Scharakteryzowano też kilka przydatnych narzędzi wspomagających proces zapewniania jakości. Szczególną uwagę poświecono zarządzaniu jakością w procesach zwinnego wytwarzaniu oprogramowania.

Plan

1. Ogólne zagadnienia zarządzania jakością.
2. Certyfikacja i modele jakości.
3. Testowanie i inne metody dbania o jakość.
4. Narzędzia wspomagające zarządzanie jakością.

3.1. OGÓLNE ZAGADNIENIA ZARZĄDZANIA JAKOŚCIĄ

Twórcy oprogramowania muszą liczyć się z rosnącymi wymaganiami klientów i partnerów biznesowych. Poza tym zapewnienie wysokiej jakości oprogramowania jest kwestią kluczową ze względu na koszty. Projekty o niskiej jakości mają zwykle słabe szanse na powodzenie zarówno na etapie tworzenia, jak i późniejszej eksploatacji i konserwacji. Koszty poprawiania, rozwijania i adoptowania niedbale stworzonego oprogramowania mogą spowodować wzrost kosztów eksploatacji nawet o 80% (Jayaswal, Patton, 2009). Szacuje się, że same testy, których przeprowadzenie jest wymagane do usunięcia błędów, pochłania ok. 40% wydatków ponoszonych na rozwój oprogramowania. Wynika z tego, że o wiele bardziej opłacalne jest unikanie błędów niż późniejsze przeprowadzanie prac nad ich eliminacją.

Jakość oprogramowania to zagadnienie wieloaspektowe. Twórcy muszą sprostać jawnym i ukrytym potrzebom klientów, oczekiwaniom komitetu sterującego, partnerów i audytorów. Ponadto, produkt musi być konkurencyjny i przede wszystkim opłacalny, stworzony w terminie i z uwzględnieniem ograniczeń budżetowych. Samą jakość można zdefiniować jako stopień spełnienia przez system stwierdzonych i przewidywanych oczekiwań klientów i innych partnerów.

Wysokiej jakości produkt powinien być funkcjonalny, bezpieczny, niezawodny i wydajny. Jednak jakość to pojęcie subiektywne i każda osoba może przyjąć inne kryteria jej oceny. Jednocześnie kryteria te mogą ewaluować i zmieniać się, co dodatkowo utrudnia osiągnięcie dostatecznego ich poziomu.

Przyczyn zawodności oprogramowania może być wiele. Do podstawowych można zaliczyć (Jayaswal, Patton, 2009):

- Brak lub niewielkie zaangażowanie kadry zarządzającej. Problemy z brakiem odpowiedniego zarządzania mogą być przyczyną nawet 85% problemów z jakością.
- Niewystarczająca interakcja z użytkownikami końcowymi. Jeśli brakuje rozmów i konsultacji z klientem często okazuje się, że wymagania użytkowników nie zostały dobrze zrozumiałe. To w oczywisty sposób prowadzi do ich niespełnienia.

Ponadto trzeba mieć też na uwadze fakt, że wymagania użytkowników zmieniają się wraz z rozwojem oprogramowania.

- Rosnąca złożoność. W trakcie rozwoju oprogramowania złożoność i stopień komplikacji oprogramowania może niekontrolowanie wzrosnąć. Rosnąca ilość dodatkowych funkcji i udogodnień komplikują system niekoniecznie czyniąc go bardziej wartościowym. Wzrost skomplikowania może jednak mieć poważny wpływ na jakość, której poziom w takich warunkach może zmaleć.
- Brak uzgodnionych kryteriów oceny jakości. Kryteria takie powinny być uzgodnione z klientem lub innymi osobami zainteresowanym powodzeniem projektu. Dzięki temu możliwe będzie wypracowanie ujednoliconego sposobu oceny systemu.
- Błędy harmonogramowania. Nieodpowiednie określenie czasu zakończenia poszczególnych zadań w projekcie może skutkować dostarczeniem produktu z uszczuplonym zakresem funkcjonalności lub ich niską jakością. Taki program będzie wymagał późniejszych poprawek, których naprawa może wywołać błędy na różnych etapach projektu.
- Brak odpowiednio rozbudowanych testów. Testowanie aplikacji jest kwestią istotną. Istnieje wiele rodzajów testów i ich dobór powinien zależeć od rodzaju i wielkości rozwijanego oprogramowania.
- Brak odpowiednich mechanizmów bezpieczeństwa. Każde oprogramowanie może być narażone na ataki. W szczególności aplikacje internetowe są narażone na różnego rodzaju zagrożenia, takie jak włamania, kradzieże tożsamości czy oszustwa finansowe. Warto więc zadbać o dobór odpowiedniego poziomu zabezpieczeń.
- Niewystarczająca motywacja. Często twórcy wolą skupić się na ciekawym wyglądzie aplikacji, jej ergonomii lub innowacyjnych funkcjach, niż na wysokiej jakości oprogramowania. Dla celów marketingowych i rynkowych takie podejście wydawać się może słuszne, jednak słaba jakość wcześniej czy później stanie się źródłem problemów.

W praktyce mówi się o trzech szkołach (podejściach) zarządzania jakością: zachodnioeuropejskiej, amerykańskiej i japońskiej.

Pierwsza z nich, zachodnioeuropejska, zakłada uzyskanie maksymalnej

zgodności produktu ze specyfikacją rozumianą jako opis wymogów technicznych i projektowych. Podejście takie koncentruje się więc na opracowaniu dokładnej dokumentacji projektowej, wprowadzeniu standaryzacji oraz opracowaniu planów testów. Jednak jakość jest definiowana przede wszystkim przez kryteria twórców programu. Wpływ klienta jest marginalny.

Podstawą podejścia amerykańskiego jest przekonanie, że produkt musi przede wszystkim spełniać wymagania klienta. Poziom jakości zależy od oceny klienta, nie zaś od opinii specjalistów IT.

Podejście japońskie z kolei wykorzystuje **zasadę Kaizen** ((jap. 改善, pol. *poprawa*, *polepszenie*), czyli nieustanne doskonalenie. Zgodnie z tą zasadą warto poprawiać nieustannie wszystko, co można jeszcze ulepszyć. Idea ta dotyczy udoskonalania procesów wytwórczych w całej firmie, i to na wszystkich jej poziomach. Dzięki temu produkt przez nią wytworzony będzie lepszej jakości niż taki, którego twórcy mniej przywiązują wagę do podnoszenia kultury jakości swojej organizacji.

Można wyróżnić kilka ogólnych zasad zarządzania jakością. Ich znajomość jest przydatna przy okazji wprowadzania polityki jakości, czyli zbioru zamierzeń, kierunków i reguł działań organizacji dotyczących zapewnienia jakości. Zasady te przedstawia tabela 3.1.

Jedną z koncepcji zarządzania, która skupia zasady wymienione w tabeli 3.1 jest **Kompleksowe Zarządzanie Jakością** (ang. *Total Quality Managemet, TQM*). Koncepcja ta jest ukierunkowana na spełnianie oczekiwań klientów dzięki zaangażowaniu pracowników w realizację polityki jakości organizacji i ciągłego jej doskonalenia. Metoda **TQM** polega na podnoszenia sprawności zespołu i zmiany sposobu myślenia. Jej podstawowe zasady to:

- stałe doskonalenie na wszystkich poziomach organizacji;
- działanie zespołowe, współpraca i lojalność w zespole;
- myślenie systemowe, podejmowanie decyzji zgodnych z polityką jakości.

Koncepcja TQM polega więc na koncentracji na zadowoleniu klienta przez ciągły rozwój produktu, ulepszanie procesu ochrony jakości tworzenia i konserwacji oprogramowania. Model TQM miał duży wpływ na rozwój oprogramowania pod koniec XX wieku. Przyczynił się też do wzrostu świadomości konieczności ulepszania procesów organizacyjnych wśród twórców oprogramowania.

Tabela 3.1. Zbiór zasad zarządzania jakością

Nr	Zasada	Opis
1.	Ukierunkowanie na klienta	Zapoznanie się i zrozumienie aktualnych i przyszłych potrzeb klienta pozwala na trafniejszą realizację ich wymagań. Zaleca się powiązanie celów organizacji z potrzebami klienta dbając przy tym o odpowiedni poziom komunikacji oraz motywację pracowników.
2.	Przywództwo	Kierownictwo określić powinno cel, kierunki rozwoju oraz kodeks wewnętrznego środowiska organizacji. Warto też zachęcać pracowników do kreatywności i samodzielności wyznaczając jednocześnie granice odpowiedzialności.
3.	Zaangażowanie	Zapewnienie jakości budowane jest przez członków organizacji na wszystkich szczeblach zatrudnienia. Każdy członek organizacji powinien rozumieć swoją rolę w zespole, znać swoje obowiązki i zakres odpowiedzialności.
4.	Podejście procesowe	Zasoby i działania w projekcie i całej organizacji powinny być traktowane jako elementy dynamicznego procesu. Istotne jest stworzenie procedur usprawniających pracę w zespole i określających niezbędne do wykonania działania. Ciągłe udoskonalanie procesu tworzenia wymaga też monitorowania funkcjonowania procedur.
5.	Podejście systemowe	Wydajność działań organizacji zwiększa się, gdy procesy powiązane są w łatwo zarządzany i rozumiany system. Takie podejście ułatwia lokalizację miejsc ryzyka w projekcie i zwiększa jego szanse na powodzenie.
6.	Ciągłe doskonalenie	W organizacji warto organizować szkolenia i inwestować w podnoszenie jakości wyrobów i procesów. Jedną z popularnych metod doskonalenia jest iteracyjny proces PDCA, czyli: Plan, Do, Check, Act (Planuj, Wykonaj, Sprawdź i Działaj).
7.	Rzetelna informacja	Wnikliwa analiza danych i informacji powinna być podstawą podejmowania decyzji w projekcie i w całej organizacji. Istotne dane dotyczące projektu należy udostępnić zainteresowanym osobom aby zminimalizować podejmowanie decyzji w warunkach niepewności.
8.	Partnerstwo dla jakości	Współpraca i partnerstwo pomiędzy organizacją a jej środowiskiem zwiększa zdolność wspólnego tworzenia. Dobre relacje z kluczowymi dostawcami i klientami powinny być one oparte o zasadę obustronnej korzyści.

Źródło: (Nawrocki, 2006)

Jakość oprogramowania, aby mogła być oceniona, musi posiadać miary lub składowe. W praktyce można je podzielić na dwa rodzaje: zewnętrzne (widoczne dla użytkownika) oraz wewnętrzne (dotyczące wewnętrznej budowy aplikacji). Składowe

zewnętrzne to przede wszystkim (McConnell, 2010):

- **poprawność** czyli brak błędów specyfikacji, projektu i samej implementacji;
- **funkcjonalność** czyli prostota obsługi i łatwość, z jaką użytkownicy uczą się korzystać z aplikacji;
- **efektywność** polegająca na jak najmniejszym wykorzystaniu zasobów sprzętowych;
- **niezawodność** rozumiana jako czas bezawaryjnej pracy aplikacji i jej zdolność do realizacji zaplanowanych funkcji;
- **integralność**, czyli zapobieganie niepoprawnemu dostępowi do funkcji i danych;
- **adaptacyjność**, czyli zakres, w jakim możliwe jest korzystanie z aplikacji bez dodatkowych modyfikacji w innych środowiskach niż dedykowane;
- **dokładność** określana jako stopień, w jakim aplikacja jest wolna od błędów, czyli czy dobrze spełnia ona swoje zadanie;
- **odporność**, czyli możliwość kontynuacji pracy w niesprzyjającym otoczeniu lub przy błędnych danych wejściowych.

Do składowych wewnętrznych można zaliczyć (McConnell, 2010):

- **łatwość konserwacji** czyli poziom modyfikowalności systemu, w tym naprawy błędów, integracji z innym oprogramowaniem czy dodania funkcjonalności;
- **elastyczność** określana jako stopień, w jakim można zmodyfikować oprogramowanie tak, aby mogło zostać użyte do innych zastosowań niż te, które były określone pierwotnie;
- **przenośność** rozumiana jako łatwość umożliwienia jego pracy w środowiskach innych niż dedykowane;
- **możliwość ponownego użycia** fragmentów aplikacji przy budowie innych systemów;
- **czytelność** kodu, czyli łatwość jego zrozumienia i poruszania się po nim;
- **testowalność**, w szczególności możliwość poddania kodu testom jednostkowym i systemowym;
- **zrozumiałość** działania systemu na różnych poziomach abstrakcji oraz jego spójność.

Granice pomiędzy wewnętrznymi i zewnętrznymi składowymi jakości w niektórych miejscach zacierają się. Niedbale napisany kod, w którym trudno wprowadza się

zmiany ma wpływ na pogorszenie się również składowych zewnętrznych. Podobnie dążenie do maksymalizacji niektórych cech może prowadzić do pogarszania się innych. Przykładowo zbytnia koncentracja na poprawności może obniżyć poziom odporności.

Kontrolowanie jakości powinno być procesem przeprowadzanym ciągle i dotyczącym wielu aspektów. Istnieje kilka metod służących poprawie jakości nie tylko pojedynczego produktu ale przede wszystkim całego procesu wytwarzania oprogramowania. Są to (McConnell, 2010):

- Wyraźne określenie celów, na których koncentrowane będą działania podnoszenia jakości. Programiści powinni znać listę charakterystyk określających te składowe jakości, które w danym projekcie są najistotniejsze.
- Zapewnienie odpowiedniej kultury organizacji, której pracownicy będą świadomi, że jakość jest traktowana priorytetowo. Wprowadzone powinny być więc odpowiednie procedury jej kontroli.
- Ustalenie strategii testowania, opracowanej w odniesieniu do wymagań, projektu i architektury systemu. Testowanie często jest uważane za podstawową metodę poprawiania jakości przedsięwzięcia.
- Określenie zbioru zasad budowy oprogramowania, w tym opracowywania wymagań, dokumentowania, tworzenia poprawnego kodu, korzystania z wzorców projektowych i testowania.
- Przeprowadzanie przez programistów nieformalnych przeglądów technicznych przed oddaniem ich do formalnego przeglądu.
- Przeprowadzanie formalnych przeglądów technicznych, czyli okresowych testów pozwalających sprawdzić, czy jakość produktu na określonym etapie tworzenia jest odpowiednia. Celem przeprowadzania takich kontroli jest wychwycenie błędów na wczesnym etapie tworzenia oprogramowania. Mogą w nich brać udział współpracownicy i klienci. Przeglądy techniczne określane są mianem „bram jakości" i zwykle przeprowadza się je po zakończeniu kolejnych etapów projektu.
- Przeprowadzanie audytu zewnętrznego, będącego rodzajem przeglądu technicznego. Audyt zewnętrzny pozwala na określenie statusu projektu oraz jego jakości. Jest on przeprowadzany przez zespół specjalistów spoza organizacji. Uwagi przekazywane są najczęściej kierownictwu projektu.

Skuteczność metod poprawy jakości bywa różna w zależności od wielkości i rodzaju projektu. W praktyce żadna z technik wykorzystywana pojedynczo nie daje więcej niż 75% skuteczności, a średnia dla wszystkich to zaledwie 40%. Przeciętnie używając wielu różnych technik można uzyskać ok. 85% skuteczności. Szczegółowe dane przedstawione są w tabeli 3.2.

Tabela 3.2. Wykrywalność defektów w praktyce

Metoda	Najniższy poziom wykrywalności	Typowy poziom wykrywalności	Najwyższy poziom wykrywalności
Nieformalne przeglądy projektu	25%	35%	40%
Formalne inspekcje projektu	45%	55%	65%
Nieformalne przeglądy kodu	20%	25%	35%
Formalne przeglądy kodu	45%	60%	70%
Modelowanie / prototypowanie	35%	65%	80%
Samodzielne śledzenie algorytmu	20%	40%	60%
Testy jednostkowe	15%	30%	50%
Testy nowej funkcji / komponentu	20%	30%	35%
Testy integracyjne	25%	35%	40%
Testy regresyjne	15%	25%	30%
Testy systemowe	25%	40%	55%
Wąskie testy beta (<10 stanowisk)	25%	35%	40%
Szerokie testy beta (>1000 stanowisk)	60%	75%	85%

Źródło: (McConnell, 2010; Jones, 1986; Jones, 1996; Shull i in., 2002)

Zostało zbadane (Meyers, 2005), że w zależności od rodzaju projektu lepsze mogą okazać się zarówno procesy bazujące na udziale człowieka jak i testy komputerowe. Jednak uważa się (Johnson 1994), że nieformalne metody testowania obejmują nie więcej niż 50-60% kodu. Dlatego zawsze warto łączyć różne metody wykrywania defektów.

Warto zwrócić też uwagę na koszty wyszukiwania i usuwania defektów i tym samym podnoszenia jakości projektu. Koszty poszczególnych metod usuwania

błędów są różne. Najbardziej ekonomiczne są te, które dają najniższy koszt usunięcia pojedynczego błędu z uwzględnieniem niezmienności pozostałych czynników. Na koszt ten wpływa również faza projektu, w którym błąd został znaleziony. Badania (Basili, Selby 1987, Ackerman i In. 1989) wykazały, że czytanie kodu i inspekcje wymagają mniejszej liczby roboczogodzin niż testowanie. Jednocześnie testowanie wymaga dodatkowej pracy w celu ustalenia przyczyn błędu. Oczywiście im wcześniej wykryty zostanie błąd, tym tańsze będzie usunięcie go.

Efektywny system zarządzania jakością powinien być połączeniem kilku metod używanych na różnych etapach rozwijania projektu. Przykładowe połączenie zawierać by mogło (McConnell, 2010): formalne inspekcje wymagań, architektury i kluczowych modułów systemu, modelowanie (lub prototypowanie), inspekcje (czytanie) kodu i testowanie uruchomieniowe. Warto pamiętać, że w tradycyjnym cyklu tworzenia oprogramowania debugowanie, refaktoryzacje i inne przebudowy programu zajmują ok. 50% czasu poświęconemu projektowi. Im lepsza jest jakość programu, tym mniej czasu trzeba poświęcić na przebudowy i poprawki.

3.2. CERTYFIKACJA I MODELE JAKOŚCI

Powstanie norm i standardów związane było z koniecznością poprawy organizacji przedsięwzięć w firmach. Takie problemy jak bałagan organizacyjny, samowola pracowników, źle dopasowane zasoby i brak jasno określonych reguł współpracy i kontroli przyczyniają się do spadku jakości działania firmy i jej produktów. Idea wprowadzenia do organizacji norm i standardów miała na celu usprawnienie jej procesów i wprowadzenie konieczności korzystania z udokumentowanych procedur. Idea ta rozpowszechniła się w latach 70-tych i 80-tych XX wieku, kiedy to wprowadzone zostały pierwsze standardy ISO 9000.

Dużą rolę w certyfikacji oprogramowania i sposobów jego wytwarzania odgrywają normy z rodziny ISO 9000 oraz norma ISO 9126. Warto także wspomnieć o modelach takich jak **CMM** (ang. *Capability Maturity Model*) czy **XPMM** (ang. *eXtreme Programming Maturity Model*).

Normy ISO 9000 dotyczą całego systemu zarządzania jakością w organizacji.

Spośród najistotniejszych norm tej rodziny można wyróżnić (Nawrocki, 2006):

- Norma ISO 9000:2005 – Podstawy i terminologia systemu zarządzania jakością,
- Norma ISO 9001:2000 – Wymagania systemu zarządzania jakością,
- Norma ISO 9004:2000 – Wytyczne doskonalenia funkcjonowania systemu zarządzania jakością,
- Norma ISO 90003:2004 – Wskazówki stosowania zapisów normy ISO 9001:2000 w odniesieniu do tworzenia oprogramowania.

Norma ISO 9000:2005 definiuje osiem zasad zarządzania jakością. Zasady te sformułowane zostały w tabeli 3.1. Ich znajomość jest istotna, ponieważ są podstawą efektywnego wdrożenia ISO.

ISO 9001:2000 jest normą, za spełnienie której można otrzymać certyfikat. Aby organizacja mogła go otrzymać, musi przejść przez dwustopniowy proces klasyfikacji. Z ośmiu rozdziałów wchodzących w skład normy ISO 9001:2000 istotną rolę odgrywa cztery (rozdziały od czwartego do ósmego). Rozdziały te to:

1. **System zarządzania jakością**. Ogólne wymagania stawiane na tym etapie dotyczą identyfikacji procesów i zależności pomiędzy nimi oraz zapewnienia zasobów do realizacji procesów monitorowania i doskonalenia procesów. Na tym etapie pod uwagę brany jest też sposób dokumentowania systemu zarządzania jakością.
2. **Odpowiedzialność kierownictwa**. Zaangażowanie najwyższego kierownictwa firmy powinna się przejawiać w działaniach potwierdzających orientację na klienta. Na tym etapie oceniane są sposoby dążenia kierownictwa do monitorowania oczekiwań klienta oraz jak najlepszego ich spełnienia. Ważna jest także odpowiednio sformułowana i aktualizowana polityka jakości. W szczególności powinna ona zawierać jasno określony cel i misję organizacji. Oceniany jest też poziom komunikacji w firmie oraz sposób sprecyzowania odpowiedzialności i uprawnień.
3. **Zarządzanie zasobami**. Konieczne jest zapewnienie zasobów potrzebnych do wdrożenia, a także późniejszego utrzymania systemu zarządzania jakością. Istotne jest też podnoszenie kwalifikacji i kompetencji pracowników. Zasoby to również infrastruktura, w tym zabudowania, wyposażenie i przestrzeń do pracy. Poziom infrastruktury organizacji również jest oceniany.

4. **Realizacja wyrobu**. Ten etap koncentruje się na procesie planowania oraz tworzenia produktu. Powinien zostać utworzony plan jakości określający wymagania dotyczące wyrobu, dokumentacja oraz sposoby i kryteria akceptacji. Pod uwagę bierze się też sposób komunikacji z klientem oraz specyfikacji wymagań. Głównym obszarem koncentracji na tym etapie jest sam proces produkcji i dostarczania produktu, w tym odpowiednie jego testowanie.
5. **Pomiary, analiza i doskonalenie**. Sprawdzanie zgodności produktu z wymaganiami jest domeną rozdziału siódmego. Ten etap koncentruje się na zgodności funkcjonującej aplikacji z opisem dostępnym w księdze jakości, a także na sprawdzaniu poziomu zadowolenia klienta. Zalecane jest używanie metod statystycznych do analizy danych. Pomiary dotyczyć powinny czterech aspektów: systemu zarządzania jakością (audity wewnętrzne), efektywność procesów systemu zarządzania jakością, samego produktu oraz oceny klienta.

Przed odniesieniem norm ISO do metodyk lekkich przedstawione zostaną szczegóły drugiej normy istotnej dla jakości oprogramowania – normy ISO 9126. Norma ta określa standardy opisu wymagań dla oprogramowania. Jej najnowsza wersja złożona jest z czterech części:

1. **Model jakości** (norma ISO/IEC 9126-1:2001). Opisuje on charakterystyki i atrybuty, które służą do definiowania wymagań niefunkcjonalnych. Czynniki te mogą też być wykorzystywane do weryfikacji oczekiwań użytkownika.
2. **Miary zewnętrzne** (norma ISO/IEC 9126-2:2003). Zestaw metryk zewnętrznych służy do pomiaru charakterystyk określających zachowanie się systemu. Miary mogą być dedykowane różnym fazom życia oprogramowania w wersji wykonywalnej, np. fazie testowania lub później implementacji.
3. **Miary wewnętrzne** (norma ISO/IEC 9126-3:2003). Wewnętrzne metryki jakości służą do pomiaru charakterystyk oprogramowania będącego w postaci niewykonywanej. Może być to np. faza projektowana lub wczesnej implementacji.
4. **Miary jakości użytkowej** (norma ISO/IEC 9126-4:2004). Zbiór metryk jakości może zostać użyty do wygenerowania raportu technicznego zawierającego rozszerzalną listę atrybutów.

Model jakości zawiera następujące charakterystyki i ich atrybuty:

- **Funkcjonalność**, czyli zestaw funkcji programu wraz z właściwościami. Atrybuty funkcjonalności to: odpowiedniość funkcji, dokładność, interoperacyjność, zgodność i bezpieczeństwo.
- **Niezawodność** rozumianą jako stabilność i zdolność do spełnienia określonych wcześniej wymagań. Jej atrybutami są dojrzałość, odporność na błędy oraz zdolność do odtworzenia.
- **Użyteczność** jako nakład pracy jaki trzeba wykonać, aby możliwa była płynna obsługa programu przez użytkowników. Do atrybutów użyteczności zalicza się łatwość zrozumienia, łatwość nauki oraz operatywność.
- **Wydajność**, czyli sposób wykorzystywania zasobów przy określonych warunkach. Do jej atrybutów można zaliczyć: wykorzystanie czasu i wykorzystanie zasobów.
- **Łatwość konserwacji**, czyli nakład pracy potrzebnej do wprowadzenia zmian. Jej atrybuty to: łatwość analizy, łatwość wprowadzania zmian, stabilność i łatwość testowania.
- **Przenośność** rozumiana jako zdolność do przenoszenia oprogramowania do innych środowisk. Do atrybutów przenośności należą: łatwość adaptacji, zgodność, łatwość instalacji oraz łatwość zastąpienia.

Standard ISO 9126 wspomaga definiowanie wymagań jakościowych oprogramowania. Podaje charakterystyki i atrybuty cech, na które twórcy oprogramowania powinni zwrócić uwagę. Lista ta może być z powodzeniem rozszerzana w zależności od potrzeb projektu. Cechy oprogramowania definiowane przez standard z powodzeniem mogą posłużyć do oceny jakości tworzonego oprogramowania.

Wprowadzenie norm ISO do organizacji wymusza porządek i sformalizowanie jej procesów i tym samym ułatwia zarządzanie. Stanowi również istotny aspekt marketingowy. Jednak w koncepcji tej istnieją także słabe strony. Przede wszystkim jeśli organizacji zależy jedynie na dokumencie certyfikacyjnym, to nikomu nie będzie zależało na prawdziwych zmianach. Z drugiej strony stosowanie ISO zmniejsza elastyczność działań i utrudnia postępowanie w nietypowych sytuacjach. Ponadto ilość dokumentacji w organizacji, która wdrożyła ISO znacznie się zwiększa. Pisanie i aktualizowanie dokumentacji zabiera sporo czasu i zasobów a nie przekłada się bezpośrednio na proces tworzenia produktu. Biurokratyzacja procesów i tworzenie formalizmów powodować może nawet zmniejszenie wydajności i opóźnienia. Problem

ten w szczególności dotyczy organizacji, które korzystają z metodyk lekkich tworzenia oprogramowania.

Trzeba jednak przyznać, że najnowszej wersje norm ISO (w szczególności ISO 9000) i innych metodyk wspomagających tworzenie jakości w organizacji są dość elastyczne. W związku z tym możliwe jest tworzenie standardów ukierunkowanych na metodyki lekkie, w tym programowanie ekstremalne. Metodyki te są dość proste. Ich działanie dąży do minimalizacji infrastruktury informatycznej, w szczególności nie wymaga skomplikowanych metod i narzędzi planowania i zarządzania. Do podstawowych cech metodyk XP zalicza się ustną komunikację, łatwość zmian i minimalizm dokumentowania. Takie podejście jest w zasadzie przeciwieństwem idei wprowadzenia norm i standardów. Jednak z drugiej strony dążenie do zapewnienia najwyższej jakości, orientacja na klienta i motywacyjne działania kierownictwa to zagadnienia wspólne dla obu podejść. Można więc zaryzykować stwierdzenie, że wprowadzenie metodyki zwinnych rozwiązałoby problemy nadmiernej biurokracji zapewniając jednocześnie jakość, do której dąży idea standardów ISO (przede wszystkim ISO 9000).

W praktyce można wyróżnić kilka zagadnień leżących na styku idei XP i ISO. Są to (Nawrocki i in., 2001):

- **Odpowiedzialność kierownictwa organizacji** jest zagadnieniem wykraczającym poza obszar metodyki XP. Idea XP dotyczy przede wszystkim projektu programistycznego.
- **Zasoby** w kontekście XP rozważane są przede wszystkim pod kątem logistyki, wielkości zespołów projektowych oraz kwestii komunikacyjnych. Z punktu widzenia ISO zasoby można rozpatrywać dwojako. Po pierwsze, programiści korzystają przede wszystkim z praktyk XP a rolę drugoplanową odgrywają procedury zapewniające działanie zgodne z ISO. Po drugie, organizacja pracy i zarządzanie zasobami powinno być nadzorowane przez odpowiedni dział organizacji.
- **Realizacja produktu** jest przedsięwzięciem, którego zasady realizacji według XP i ISO pokrywają się. Instrukcje i procedury jakości mogą opisywać praktyki XP, takie jak np. programowanie w parach lub przeprowadzanie inspekcji.

- **Pomiary** to kolejny aspekt tworzenia oprogramowania, który godzi podejścia XP i ISO. Ze względu na koszty metodyka XP zaleca minimalizowanie ilości pomiarów na początku projektu. Zamiast tego zakłada ona koncentrację na przeglądaniu i tworzeniu nowego kodu. Norma ISO natomiast zaleca gromadzenie informacji na temat kosztu realizacji zdefiniowanych wymagań. Powinna też zostać wyznaczona osoba, która zajęłaby się tym zadaniem odciążając jednocześnie programistów pracujących zgodnie z założeniami XP.
- **Procedury i zapisy** to podstawa założeń normy ISO. Stosować się je powinno przede wszystkim w przypadkach zakłóceń realizacji procesu. Takie podejście jest również doceniane przez metodykę XP, która zaleca opracowanie przypadków testowych dla każdego błędu znalezionego w oprogramowaniu.

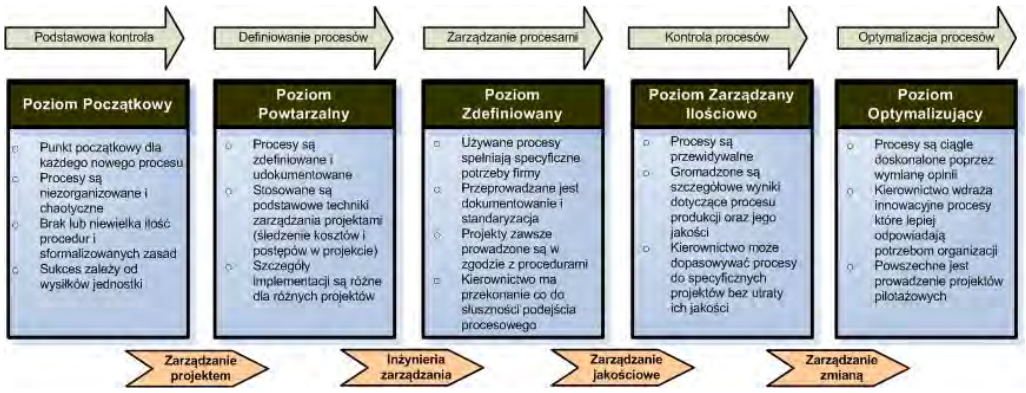
Z powyższego wynika, że możliwe jest połączenie wydawać by się mogło przeciwstawnych idei tworzenia oprogramowania. W podobny sposób można przeanalizować możliwości współistnienia metodyk zwinnych z modelami odniesienia dla oceny jakości procesu tworzenia oprogramowania. Do najpopularniejszych modeli tego typu należy Model Dojrzałości Wytwórczej – CMM (ang. *Capability Maturity Model*) i jego rozszerzona wersja - **CMMI** (ang. *Capability Maturity Model Integration*).

Modele CMM/CMMI dotyczą takich dyscyplin jak (Nawrocki, 2006) inżynieria systemów, inżynieria oprogramowania, zintegrowany rozwój produktu i procesu oraz zlecenia i dostawy. Model obejmuje pięć hierarchicznie ułożonych poziomów. Przedstawia je rysunek 3.1. Dodatkowo każdy (z wyjątkiem pierwszego) poziom posiada zdefiniowane tzw. Kluczowe obszary procesu (ang. *Key Process Areas)* charakteryzujące dojrzałość procesu tworzenia oprogramowania. Poziomy CMM to (Nawrocki, 2006):

1. Poziom pierwszy – **początkowy**, to poziom, na którym znajdują się organizacje które nie spełniają wymagań stawianych na wyższych poziomach. Brak kluczowych obszarów procesu.
2. Poziom drugi – **powtarzalny** (dla CMMI - zarządzany), to poziom organizacji spełniających podstawowe praktyki dotyczące zarządzania przedsięwzięciem informatycznym. Kluczowe obszary procesu to: *Zarządzanie wymaganiami, Planowanie przedsięwzięcia programistycznego, Nadzór i kontrola, Zarządzenie podwykonawcami, Zapewnianie jakości, Zarządzenie konfiguracjami.*

3. Poziom trzeci – **zdefiniowany**, to poziom związany z bardziej zaawansowanymi praktykami dotyczącymi całej organizacji. Kluczowe obszary procesu: Zarządzanie wymaganiami, Planowanie przedsięwzięcia programistycznego, Nadzór i kontrola, Zarządzenie podwykonawcami, Zapewnianie jakości, Zarządzenie konfiguracjami.
4. Poziom czwarty – **zarządzany ilościowo**, to poziom stosowania zaawansowanych praktyk analizy danych dotyczących efektywności i jakości procesów tworzenia oprogramowania. Kluczowe obszary procesu: Ilościowe zarządzanie procesem, Zarządzanie jakością oprogramowania.
5. Poziom piąty – **optymalizujący**, klasyfikuje organizacje, które są w stanie systematycznie przygotowywać się i wprowadzać zmiany. Kluczowe obszary procesu: Zapobieganie defektom, Zarządzanie zmianą technologii, Zarządzanie zmianą procesu.

Warto zauważyć, że organizacje znajdujące się na poziomie początkowym nie dysponują środowiskiem sprzyjającym tworzenia wysokiej jakości oprogramowania. Typowe problemy takich firm to słaba organizacja, niezmotywowani pracownicy, brak pracy zespołowej. W organizacji takiej brak jest jakichkolwiek procesów.



Rys. 3.1. Poziomy modelu CMM

Źródło: (http://www.computerworld.com/s/article/print/99159/CMMI)

Porównując ideę CMM z metodyką XP można znaleźć podobieństwa zwłaszcza na drugim poziomie. Takie wnioski płyną z analiz porównawczych obu metodyk (tab. 3.3).

Tabela 3.3. Zgodność CMM z XP

Poziom	Kluczowy obszar procesu	Zgodność z XP według twórcy CMM	Zgodność z XP według niezależnego eksperta
2	Zarządzanie wymaganiami	++	+-
2	Planowanie przedsięwzięcia programistycznego	++	+
2	Nadzór i kontrola	++	+-
2	Zarządzenie podwykonawcami	-	-
2	Zapewnianie jakości	+	+/-
2	Zarządzenie konfiguracjami	+	+/-
3	Ukierunkowanie na proces organizacji	+	+/-
3	Definicja procesu organizacji	+	-
3	Program szkoleń	-	-
3	Zintegrowane zarządzenie oprogramowaniem	-	-
3	Inżynieria produktu programowego	++	+/-
3	Współpraca między grupami	++	+/-
3	Przeglądy	++	+
4	Ilościowe zarządzanie procesem	-	-
4	Zarządzanie jakością oprogramowania	-	-
5	Zapobieganie defektom	+	+/-
5	Zarządzanie zmianą technologii	-	-
5	Zarządzanie zmianą procesu	-	-

Źródło: (Paulk, 2001; Nawrocki i in., 2001)

W rzeczywistości podstawową różnicą pomiędzy obydwoma podejściami jest konieczność dokumentowania w CMM i pomijanie go w XP. Programowanie ekstremalne neguje też potrzebę gromadzenia danych historycznych w inny sposób niż poprzez budowę doświadczenia. Metodyka CMM natomiast zaleca dokumentowanie wszystkich wymagań, planów, procesów oraz zmian. Aby więc pogodzić oba podejścia, trzeba zgodzić się na pewne kompromisy. J. Nawrocki (Nawrocki i in., 2001) wskazuje, że zmiana w XP mogłaby dotyczyć wprowadzeniu procesów usprawniających pielęgnację oprogramowania oraz rozstrzygania konfliktów

na etapie definiowania wymagań. W modelu CMM natomiast warto by było zrezygnować z części dokumentacji takiej jak np. Planu rozwoju oprogramowania.

Wartym uwagi modelem ukierunkowanym na metodyki lekkie tworzenia oprogramowania jest **XPMM** (Nawrocki, 2001a). Może być on wykorzystany do oceny stopnia, w jakim organizacja stosuje praktyki XP. Model ten składa się z czterech następujących po sobie poziomów, z których każdy (poza pierwszym) opisany jest tzw. kluczowymi obszarami procesu. Poziomy te to (Nawrocki, 2001):

1. Poziom pierwszy – niezgodny z XP, to poziom, na którym znajdują się projekty tworzone niezgodnie z praktykami programowania ekstremalnego. Brak kluczowych obszarów procesu.
2. Poziom drugi – początkowy, uwzględnia kwestie relacji z klientami oraz zapewniania jakości oprogramowania. Kluczowe obszary procesu: stosowanie gry planistycznej, wykorzystywanie opowieści użytkownika, tworzenie harmonogramu w oparciu o planowanie wydań, śledzenie postępu prac.
3. Poziom trzeci – zaawansowany, to poziom skupiający się na programowaniu w parach z uwzględnieniem zdefiniowania zakresu odpowiedzialności każdego programisty. Kluczowe obszary procesu: rotacja struktury zespołu, częste spotkania klienta z twórcami programu, uwzględnianie standardów tworzenia kodu, współwłasność kodu, korzystanie z systemu kontroli wersji, automatyzacja testów.
4. Poziom czwarty – dojrzały, to poziom odpowiedzialny za monitorowanie satysfakcji klienta oraz twórców oprogramowania. Kluczowe obszary procesu: obecność klienta podczas procesu tworzenia oprogramowania, brak nadgodzin, używanie testów jednostkowych, zadowolenie klienta.

Jak widać istnieje wiele podejść do procesu zapewniania jakości tworzenia oprogramowania. To, co skupia wszystkie przedstawione modele i metodyki to ukierunkowanie na klienta i dążenie do uzyskania możliwie najwyższego poziomu jego zadowolenia. Ważne jest, aby wprowadzanie do organizacji nowej metodyki było dobrze przemyślane i umotywowane. Tylko takie podejście zapewni faktyczną zmianę na lepsze.

3.3. TESTOWANIE I INNE METODY DBANIA O JAKOŚĆ

Jedną z podstawowych zasad programowania zwinnego jest **wytwarzanie sterowane testami** (ang. *Test-Driven Development, TDD*). Metoda ta wymaga takiego napisania kodu, by możliwe było zaliczenie testu jednostkowego. Tak więc praca nad kodem rozpoczyna się od utworzenia odpowiedniego testu jednostkowego. Kod pisany jest w drugiej kolejności. Istnieją trzy proste zasady TDD (Martin i Martin, 2007):

1. Nie należy rozpoczynać pracy z kodem dopóki nie zostanie utworzony odpowiedni test jednostkowy.
2. Nie należy pisać testów, które nie są niezbędne do późniejszego tworzenia kodu.
3. Należy pisać tylko tyle kodu, ile jest wymagane do przejścia istniejących testów jednostkowych.

Taki sposób tworzenia oprogramowania charakteryzuje się pracą w krótkich cyklach (od momentu utworzenia pustego testu do wypełnienia go odpowiednim kodem). Przypadki testowe ewoluują wraz z rozwojem kodu co sprawia, że dla każdej funkcji programu zawsze istnieje aktualny test weryfikujący jej poprawność. Podczas wprowadzania zmian można więc w łatwy sposób sprawdzić, czy nie będą one miały negatywnego wpływu na pozostałe komponenty aplikacji. Modyfikacje można więc przeprowadzać w łatwy i bezpieczny sposób. Ponadto zaletą tak tworzonego oprogramowania jest prostota, którą wymusza spełnienie wyżej wymienionych zasad.

Warto też zauważyć, że podejście takie zachęca do tworzenia modułów, których elementy mogą być testowane niezależnie od siebie. Wzajemne zależności są więc minimalizowane, co ułatwia i przyspiesza dalszy rozwój oprogramowania.

Kolejną istotną zaletą metody TDD jest możliwość potraktowania utworzonych testów jako dokumentację aplikacji. Testy zawierają informację o tym jak wywołać daną funkcję lub utworzyć obiekt. Mogą być więc traktowane jako zbiór informacji przydatnych dla innych programistów którzy zamierzają pracować z wcześniej utworzonym kodem.

Traktując o **testach jednostkowych** konieczne wydaje się przytoczenie ich definicji. Test ten jest uruchomieniem ukończonego elementu programu (np. klasy, funkcji lub podprogramu) z jednoczesną weryfikacją poprawności jego działania. Weryfikacja taka polega na porównaniu wyników wykonania testowanego elementu

z ich oczekiwanymi rezultatami. Wyniki takie mogą być np. zwróconymi wartościami, stanami obiektów czy wyrzuconymi wyjątkami.

Pisanie testów jednostkowych przed przystąpieniem do właściwego kodowania pozytywnie wpływa na kilka aspektów tworzenia oprogramowania. Po pierwsze, prowadzi do wykrycia obszarów wymagających sprecyzowania. Po drugie, może okazać się konieczne wydzielenie elementów kodu do odrębnych modułów. To z kolei prowadzi do eliminacji sztywnych związków i zwiększenia poziomu elastyczności oprogramowania.

Przykład wykorzystania testów jednostkowych dla klasy przedstawiają Listingi 3.1 oraz 3.2. Listing 3.1 przedstawia klasę PodstawoweOperacje posiadającą dwie metody: Dodaj oraz Pomnoz. Listing 3.2 zawiera przykład poprawnie uruchamianego testu dla klasy PodstawoweOperacje. Treść testu można interpretować następująco. Po zdefiniowaniu obiektu klasy określone są przykładowe liczby, dla których wykonywane będą operacje. Utworzone są dwie zmienne przechowujące odpowiednio wynik operacji dodawania i mnożenia. Ostatnie dwie zmienne zawierają wartości oczekiwane testu. Następnie wykonywane są operacje (operacje.Dodaj, operacje.Pomnoz), których wyniki porównuje z wartościami oczekiwanymi. W zależności od wyniku test uznawany jest za poprawny lub też wyświetlany jest komunikat o błędzie.

Listing 3.1. Przykład testowanej klasy

```
public class PodstawoweOperacje {
      public int Dodaj(int liczbaA, int liczbaB)
      {
            return liczbaA + liczbaB;
      }
      public int Pomnoz(int liczbaA, int liczbaB)
      {
            return liczbaA * liczbaB;
      }
   }
```

Listing 3.2. Przykład testu jednostkowego dla klasy z Listingu 3.1

```
[TestMethod()]
public void WykonajTest() {
      PodstawoweOperacje operacje = new PodstawoweOperacje();
      int liczbaA = 2;
      int liczbaB = 3;
      int wynikDodawania = 0;
      int wynikMnozenia = 0;
      int oczekiwaneDodawanie = 5;
      int oczekiwaneMnozenie = = 6;

      wynikDodawania = operacje.Dodaj(liczbaA, liczbaB);
      wynikMnozenia = operacje.Pomnoz(liczbaA, liczbaB);

      Assert.AreEqual(oczekiwaneDodawanie, wynikDodawania);
      Assert.Inconclusive("Błędny wynik dodawania.");

      Assert.AreEqual(oczekiwaneMnozenie, wynikMnozenia);
      Assert.Inconclusive("Błędny wynik mnożenia.");
   }
```

Same testy jednostkowe działają na poziomie małych elementów systemu. Dlatego ich przeznaczeniem nie jest weryfikacja poprawności funkcjonowania całej aplikacji. Można powiedzieć, że są one przykładem testów tzw. białej skrzynki, czyli testami uwzględniającymi wewnętrzną strukturę testowanego modułu (Martin i Martin, 2007). Testami czarnej skrzynki (niezależnymi od wewnętrznej struktury aplikacji) można określić testy akceptacyjne, które są przeprowadzane na poziomie całej aplikacji.

Testy akceptacyjne są tworzone przez osoby niezaznajomione z wewnętrznymi mechanizmami aplikacji (np. analityków biznesowych lub klientów). Służą one klientowi do sprawdzenia, w jakim stopniu spełnione zostały jego wymagania. Testy takie są zautomatyzowane. Zwykle tworzy się je w prostych w obsłudze, choć jednocześnie wyspecjalizowanych językach programowania. To one stanowią ostateczną dokumentację wymagań oraz sposobu funkcjonowania aplikacji. Na ich podstawie programiści mogą określić implementację określonych elementów systemu. Sposób pisania tych testów ma wpływ na architekturę systemu. Aby testowanie było możliwe, system musi być w odpowiedni sposób podzielony. Przykładowo, interfejs

użytkownika powinien być oddzielony od reguł biznesowych.

Nie zaleca się ręcznego przeprowadzania testów akceptacyjnych. Takie podejście zmniejsza świadomość konieczności dzielenia systemu na luźno powiązane komponenty. Poza tym dodatkową zaletą testów automatycznych jest fakt, że założenie konieczności ich przeprowadzenia od początku wpływa na sposób budowy oprogramowania. Same testy powinny być utworzone jeszcze przed powstaniem oprogramowania, więc nawet samo ich tworzenie ma pozytywny wpływ na architekturę programu.

Testy jednostkowe i akceptacyjne to tylko niektóre z dostępnych sposobów testowania. Do innych należą (McConnell, 2010):

- **Testowanie komponentów**, czyli uruchamianie elementów takich jak klasy, pakiety czy podprogramy tworzone przez grupy lub całe zespoły programistów. Elementy takie testowane są pojedynczo, niezależnie od pozostałej części aplikacji.
- **Testowanie integracyjne** – uruchamianie przynajmniej dwóch elementów (klas, pakietów, komponentów) utworzonych przez grupy lub całe zespoły programistów. Testy takie rozpoczyna się gdy tylko są już gotowe pierwsze klasy i przeprowadza się je aż do chwili ukończenia projektu.
- **Testowanie regresyjne**, polegające na powtarzaniu wcześniej przeprowadzonych testów aby upewnić się, że dodanie nowych elementów programu nie przyczyniło się do powstania błędów w działaniu elementów już istniejących.
- **Testowanie systemowe**, czyli uruchamianie ukończonego oprogramowania z jego finalną konfiguracją. Testowane są aspekty bezpieczeństwa, wydajność, współpracę z zewnętrznymi zasobami (np. sprzętem) oraz poziom integracji z innymi programami.

Wymienione testy są wykonywane przede wszystkim przez programistów. Istnieje też zestaw testów wykonywanych przez specjalistycznych testerów. Są to (McConnell, 2010): testy beta, testy wydajnościowe i obciążeniowe, testy konfiguracji, testy platformowe oraz testy funkcjonalnościowe.

Testowanie jest uważane za ważny element systemu zapewniania jakości. Nie powinien być on jednak jedynym elementem. Testowanie zabiera sporo czasu, a jego

skuteczność w wyszukiwaniu błędów rzadko przekracza 60%. Wiele testów pozostawia się do przeprowadzenia samym programistom. Jednak w niektórych aspektach może okazać się to trudne. Celem pracy programisty przede wszystkim tworzenie kodu, co nie pokrywa się z celem przeprowadzania testów. Jednocześnie dobrze przeprowadzone testowanie wymaga założenia o istnieniu błędów. Programista natomiast liczy na jak najniższą liczbę błędów, co w konsekwencji może spowodować ich przeoczenie. Ponadto często zdarza się, że programista pomija mniej typowe błędy. Ponadto wielu programistów przeprowadza tzw. „czyste testy" sprawdzając przede wszystkim, czy kod w ogóle działa. Bardziej doświadczone organizacje wskazują jednak na konieczność tworzenia tzw. „brudnych testów", które polegają na sprawdzaniu różnych możliwości wywołania zakłóceń w pracy programu. Dobrą praktyką jest także używanie list kontrolnych dla typowych, przewidywanych lub często pojawiających się błędów.

Dodanie nowego elementu (np. klasy) do budowanego programu powinno być poprzedzone sprawdzeniem jego poprawności i wykonaniem założonych testów. Nieodpowiednie przetestowanie elementu przed jego dodaniem do aplikacji może spowodować trudne do późniejszej naprawy błędy. Podobnie po utworzeniu kilku funkcji powinno się przetestować każdą z nich oddzielnie. Testami powinien być pokryty cały kod programu. Ich tworzenie należy rozpocząć od testów przepływu danych a następnie dodać testy dotyczące elementów specyfikacji wymagań oraz projektu. Aby zminimalizować ryzyko przeoczenia błędów powinno się planować testowanie modułu jeszcze przed jego napisaniem, najlepiej na etapie projektowania. Można podać wiele argumentów przemawiających za tworzeniem testów na początku pracy. Są to (McConnell, 2010):

- wcześniej napisane testy pozwalają szybciej wykryć błędy,
- tworzenie testów przed rozpoczęciem programowania daje okazję do wnikliwszej analizy wymagań a to z kolei poprawia jakość pracy,
- wczesne pisanie testów daje szansę zauważenia potencjalnych problemów ze specyfikacją wymagań,
- zawsze istnieje możliwość późniejszego dopisania dodatkowych testów.

Niezależnie od rodzaju i ilości przeprowadzonych testów nie jest możliwe pełne sprawdzenie programu we wszystkich możliwych warunkach oraz dla wszystkich

dostępnych kombinacji danych wejściowych. Istnieje jednak kilka przydatnych, praktycznych technik testowania ułatwiających pracę z programem. Są to (McConnell, 2010):

- **Niepełne testowanie**. Polega na eliminacji testów nic nie wnoszących, powielających działanie wcześniej przeprowadzonych sprawdzań. Testowanie przeprowadzać należy na reprezentatywnych przypadkach różnych grup danych wejściowych.
- **Pełne testowanie bazowe**. Metoda polega na testowaniu z pokryciem całości logiki projektu przy użyciu jak najmniejszej liczby testów. Minimalna liczba testów wyznaczana jest na podstawie ilości wszystkich ścieżek. W szczególności trzeba więc zwrócić uwagę na słowa kluczowe takie jak `if`, `while`, `repeat`, `for`, `and`, `or`, `case`, `default`. Im bardziej skomplikowana procedura, tym więcej ścieżek i przypadków testowych. Łatwość testowania więc motywuje to uproszczania procedur i wyrażeń logicznych.
- **Testowanie przepływu danych**. Metoda ta kontroluje deklaracje i inicjacje danych. Składa się z dwóch rodzajów testów. Pierwszy polega na sprawdzeniu nietypowych schematów definiowania i używania zmiennej (np. definiowanie zmiennych tuż przed wyjściem z procedury, dwukrotne definiowanie lub zwalnianie tej samej zmiennej, definiowanie zmiennej po jej użyciu). Drugim rodzajem testu jest sprawdzenie wszystkich możliwych ścieżek typu zdefiniowanie-używanie zmiennej. Przykładem mogą być ścieżki z Listingu 3.3. Można wyłonić tu cztery testy: test1 (Warunek1=True, Warunek2=True), test2 (Warunek1=False, Warunek2= False), test3 (Warunek1=True, Warunek2=False), test4 (Warunek1=False, Warunek2=True). Warto zwrócić uwagę, że test1 i test2 pokrywają pełne testowanie bazowe.
- **Dzielenie według równoważności**. Technika ta polega na eliminacji testów, których użycie prowadzi do znalezienia tych samych błędów, które zostały wykryte przez inny test. Metoda przydatna jest podczas badania programu z punktu widzenia specyfikacji (nie kodu).
- **Przewidywanie błędów**. Doświadczeni programiści potrafiący przewidywać potencjalne miejsca wystąpienia błędów tworzą często dodatkowe testy oparte o własne przemyślenia. Tą nieformalną technikę można powiązać z ideą

tworzenia listy typowych błędów, która przydaje się podczas testowania nowego kodu.

- **Analiza wartości granicznych**. Metoda ta zakłada tworzenie testów sprawdzających krańce przedziałów. Zastąpienie wyrażenia >= znakiem > to przykład typowego błędu wartości granicznej. Bardziej złożonym przykładem jest uwzględnienie możliwości uzyskania wyjątkowo dużego wyniku operacji mnożenia dwóch zmiennych lub też przekazania do procedury nietypowo długiego ciągu znaków.
- **Klasy złych danych**. Metoda polega na sprawdzeniu obecności niewłaściwych danych, takich jak zły rozmiar danych, niewłaściwa ilość danych lub zły ich rodzaj.
- **Klasy dobrych danych**. Zdarza się, że testowane są jedynie złe dane, podczas gdy zapomina się o sprawdzeniu działania programu dla danych typowych. Testy dla klasy dobrych danych powinny obejmować przypadki przeciętne, minimalną oraz maksymalną normalną konfigurację a także zgodność ze starymi danymi (w przypadku np. wprowadzania nowej wersji programu).

Listing 3.3. Przykład testowania przepływu danych

```
if (Warunek1) {
   x=a;
}
else {
   x=b;
}
if (Warunek2) {
   y=x*2;
}
else {
   y=x+5;
}
```

Podstawą efektywnego testowania jest ich odpowiednie planowanie, najlepiej jeszcze w fazie początkowej całego projektu. Ważne jest, aby uwzględnić w projekcie czas, którego testy będą wymagać. Aby zapewnić lepszą jakość oprogramowania konieczne jest także zrozumienie istotności poprawnego wykonania testów. Kolejnym ważnym

aspektem jest powtarzalność testów. Jeśli warunek ten nie jest spełniony, nie będzie możliwe wykonanie pracy nad usprawnieniem ani zautomatyzowaniem testów. Ponadto powtarzanie testów jest konieczne w przypadku przeprowadzania testów regresyjnych, które są szansą na znalezienie ukrytych błędów spowodowanych rozbudowaniem programu o kolejny element lub dodania modyfikacji. Testy regresyjne muszą być takie same, aby mogły wskazać błąd. Systematyczne powtarzanie testów jest więc podstawą tworzenia dobrej jakości oprogramowania.

Wielokrotnie wykonywane testy są wydajne, jeśli podda się je automatyzacji. Do zalet automatyzacji zalicza się łatwą dostępność raz zautomatyzowanych testów, mniejsze prawdopodobieństwo popełnienia błędu w teście, możliwość częstego uruchamiania, duże szanse na wczesne wykrycie błędu.

Testowanie stanowi jedynie element programu zwiększania jakości i samo z siebie nie zapewnia eliminacji błędów. Jednak im wcześniej błąd zostanie dostrzeżony, tym jego skutki będą mniej kosztowne. Warto poświęcać czas na tworzenie i usprawnianie różnego rodzaju testów. W praktyce najbardziej efektywną metodą zapewnienia skutecznego testowania jest przeprowadzanie systematycznych pomiarów oraz ciągłe doskonalenie procesu przeprowadzania testów. Warto jest do tego celu gromadzić bazę wiedzy zawierającą porady i wskazówki dotyczące najbardziej typowych błędów.

3.4. NARZĘDZIA WSPOMAGAJĄCE ZARZĄDZANIE JAKOŚCIĄ

Istnieje wiele narzędzi, które można uznać za narzędzia wspomagające dbanie o jakość. Odpowiednie narzędzia można dobrać do potrzeb członków organizacji na różnych jej szczeblach.

Pierwszą grupą narzędzi dedykowaną programistom oraz grupom testującym są narzędzia wspomagające testowanie. Do tej grupy należą między innymi **rusztowania** dla testów, narzędzia porównywania danych, generatory danych testowych, monitory pokrycia, narzędzia zakłócające pracę systemu oraz bazy błędów.

„Rusztowania” buduje się aby ułatwić testowanie pojedynczych klas w projekcie. Można wskazać trzy podstawowe typy rusztowań:

- Klasy puste (zastępcze) to klasy z pustymi procedurami z których korzysta klasa poddawana testowaniu. Klasy takie można wykorzystać do sprawdzania przekazywanych danych, testowania poprawności parametrów, symulowania działania różnych operacji (np. zajmując odpowiednio dużo taktów zegara procesora) lub odgrywania roli uproszczonej wersji prawdziwego obiektu.
- Jarzmo testów, czyli dodatkowo utworzone procedury wywołujące procedurę testowaną. Procedury takie służą do jednokrotnego lub wielokrotnego wywoływania obiektu z określonym zbiorem parametrów wejściowych. Można je także wykorzystać do pobierania danych z różnych źródeł (np. plików, linii poleceń, innych procedur) i wywoływania obiektu testowanej klasy dla tych właśnie danych.
- Pliki zastępcze, będące uproszczoną wersją prawdziwych plików, zawierającą składniki o tych samych typach danych. Praca na niewielkich plikach zastępczych ułatwia szukanie błędów, ponieważ ich zawartość jest zwykle bardziej przejrzysta niż zawartość oryginalnych plików.

Budowa rusztowań pochłania czas, jednak w praktyce testowanie z ich użyciem jest dużo szybsze ponieważ koncentruje się jedynie na istotnych z punktu widzenia scenariusza testowego zadań. Takie podejście przydatne jest szczególnie w pracy z rozbudowanymi algorytmami, których działanie trwa zwykle sporą ilość czasu.

Innym rodzajem narzędzia przydatnego przede wszystkim podczas testów regresyjnych jest narzędzie porównywania danych. Testy regresyjne są uruchamiane wiele razy, dlatego zautomatyzowanie sprawdzania uzyskanych wyników znacznie oszczędza czas. Nawet najprostsze sposoby porównywania danych, choćby zapisywanie ich do pliku i sprawdzanie różnic korzystając z narzędzi typu diff, ułatwiają pracę i usprawniają wyszukiwanie defektów.

Wiele aplikacji tworzonych jest pod kątem zarządzania lub przetwarzania dużej ilości danych w różnych postaciach (np. plikach). W takich przypadkach warto skorzystać z generatorów danych testowych. Właściwie zaprojektowany generator pozwoli uzyskać dużą ilość danych różnych typów. To znacznie ułatwia proces testowania i pozwala na sprawdzenie przypadków, które mogłyby zostać pominięte podczas testowania ręcznego.

Badania pokazują (Wiegers, 2002), że narzędzia śledzące, takie jak **monitory**

pokrycia, umożliwiają znaczny wzrost poziomu pokrycia kodu testami. Narzędzia takie automatycznie sprawdzają, które fragmenty kodu nie zostały sprawdzone i wskazują miejsca, gdzie testy należy uzupełnić.

Przydatnym w testowaniu i diagnozowaniu typem narzędzi są **rejestratory danych**. W wersji roboczej aplikacji rejestrowanie akcji w systemie oraz stanu w przypadku awarii pomaga zdiagnozować problem i znaleźć odpowiedzialne za niego defekty aplikacji.

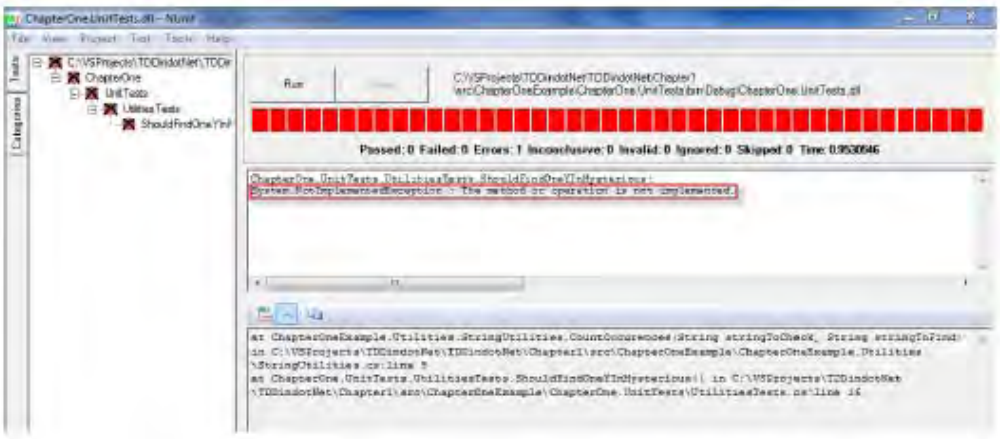
Podczas przeprowadzania testów systemowych warto skorzystać z **narzędzi zakłócających pracę** systemu. Pomogą one zamodelować okoliczności, w których aplikacja będzie musiała działać, a które nie zostały sprawdzone w warunkach produkcyjnych. Funkcje tych narzędzi mogą być różne:

- Wypełnianie pamięci, które zapewni, że wszystkie zmienne programu zostaną zainicjowane. Pamięć wypełniana będzie różnymi wartościami co pozwoli na sprawdzenie, czy któraś z nich nie wywoła błędu.
- „Wstrząsanie" pamięcią zmieniające organizację pamięci w trakcie pracy programu. Jeśli oprogramowanie korzysta z adresowania bezwzględnego, takie testowanie to wykryje.
- Selektywne zakłócanie pracy pamięci symulujące ograniczanie pamięci przez inne programy oraz wystąpienie problemów z przydziałem pamięci. Pozwala sprawdzić zachowanie oprogramowania w przypadku wystąpienia problemów z przydziałem pamięci.
- Sprawdzanie dostępu do pamięci kontrolujące operacje wskaźnikowe.

Przydatnym narzędziem wspomagającym testowanie są **bazy błędów** zawierające informacje o typowych, często spotykanych błędach. Bazę taką tworzy się w trakcie tworzenia wielu aplikacji. Takie gromadzenie danych z czasem pozwoli na utworzenie cennego zbioru porad i dobrych praktyk dotyczących testowania. Dane, które warto gromadzić w takiej bazie to opis zaistniałego błędu, kroki niezbędne do jego naprawy, powiązane problemy, liczba wierszy, które należy poprawić i czas, jaki zajmuje usuwanie defektu. Dodatkowo błędy warto klasyfikować dodając do nich informacje o randze problemu i jego wpływie na działanie całej aplikacji.

Istnieje wiele narzędzi wspomagających różne rodzaje testów. Przykładowym narzędziem wspomagającym testy webowe jest Watin. Pozwala on nagrywać

i odtwarzać sekwencje zdarzeń na stronie WWW. Popularne narzędzia obsługi testów jednostkowych są JUnit, NUnit, Rihno, TestNG. Rys. 3.2 przedstawia przykładowy ekran narzędzia NUnit.

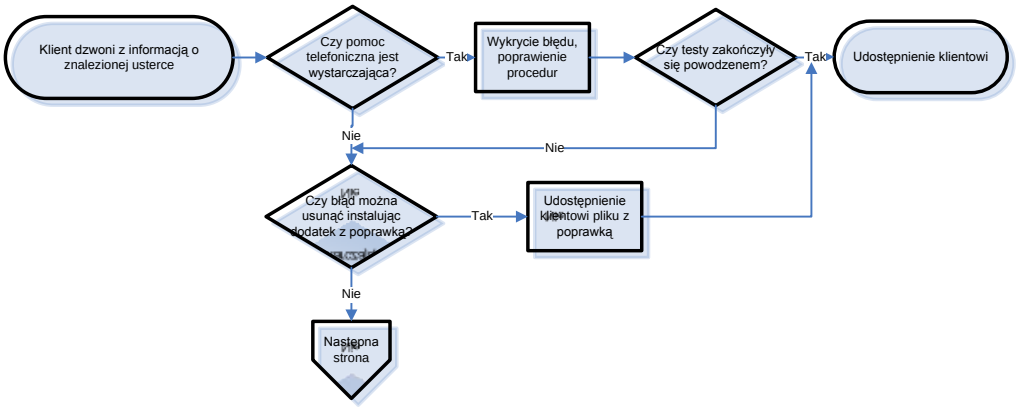


Rys. 3.2. Przykładowy ekran narzędzia NUnit
Źródło: (Bender, MCWherter, 2011)

Poza narzędziami testowania oprogramowania istnieje także zestaw narzędzi dedykowanych procesowi zarządzania jakością całego projektu. Zestaw ten wywodzi się z japońskiej szkoły zarządzania jakością. Zawiera siedem prostych w użyciu narzędzi, z których część to narzędzia statystyczne. Są to (Jayaswal, Patton, 2009): diagram przebiegu, diagram Pareto, diagram przyczynowo-skutkowy, diagram rozproszenia, arkusz kontrolny, histogram i karta kontrolna.

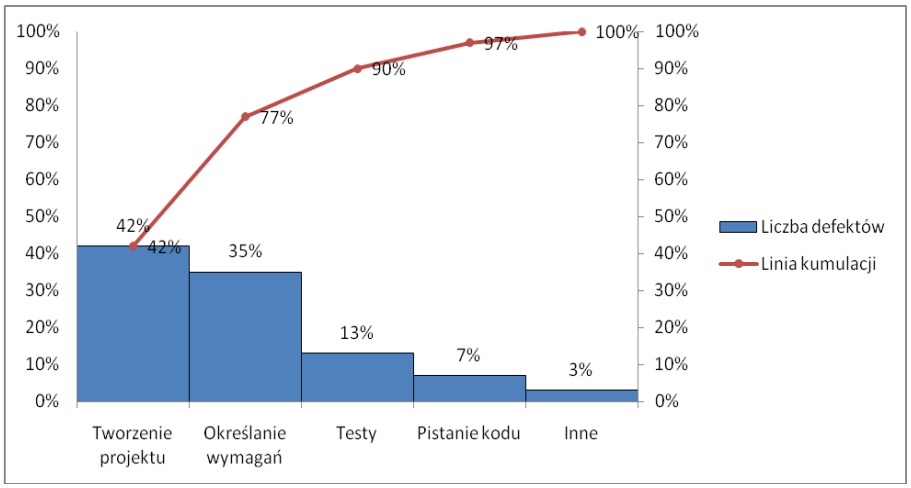
Diagram przebiegu, określany także jako mapa procesu, to graficzny sposób przedstawienia procesów projektu. Diagram przedstawia operacje takie jak operacje, pętle decyzyjne czy kontrole. Szczegółowo narysowany diagram przedstawia aktywności, których analiza może wykazać nieprawidłowości jeszcze na etapie projektu. Oczywiście diagram jest też narzędziem do komunikacji i służy do definiowania i usprawniania procesu rozwoju oprogramowania. Można utworzyć dwa rodzaje diagramów przebiegu. Pierwszy z nich, diagram wysokiego poziomu, to diagram przedstawiający ogólne, szerokie spojrzenie na cały proces. Przedstawia on podstawowe etapy procesu i jego stany pośrednie. Z punktu widzenia zarządzania

daje to możliwość grupowania osób zaangażowanych w projekt. Szczegółowy diagram przebiegu dokładnie obrazuje wybrany proces i obejmuje jego aktywności podrzędne. Analiza tego diagramu daje możliwość wprowadzenia dodatkowych usprawnień oraz eliminacji zbytecznych działań w projekcie. Przykładowy szczegółowy diagram przebiegu przedstawia rys. 3.3.



Rys. 3.3. Przykładowy szczegółowy diagram przebiegu
Źródło: (Jayaswal, Patton, 2009)

Diagram Pareto to druga technika z zestawu narzędzi zarządzania jakością. Jest to wykres słupkowy z linią kumulacji. Umożliwia on wyizolowanie najbardziej wpływowych obszarów dla analizowanego zagadnienia. Diagram wykorzystywany jest często w zarządzaniu jakością, ponieważ łatwo jest przedstawić na nim kluczowe obszary, które warto uwzględnić w zadaniach takich jak podnoszenie jakości, dotrzymanie terminowości, bilansowania kosztów, minimalizowanie liczby błędów czy poprawa zabezpieczeń. Diagram tworzy się na podstawie analiz i pomiarów lub szacunków i przewidywań. Rysunek 3.4 przedstawia przykładowy diagram Pareto przedstawiający przyczyny powstawania błędów na różnych etapach rozwoju oprogramowania. Jak wynika z badań (Jayaswal, Patton, 2009), najczęstszym źródłem błędów są niedoskonałości procesu projektowania. Błędy takie zwykle uwidaczniają się w fazach późniejszych i stanowią poważne źródło utraty kosztów. Wnioski są oczywiście takie, że warto koncentrować się na wyszukiwaniu i usuwaniu defektów już we wczesnych fazach projektowania.

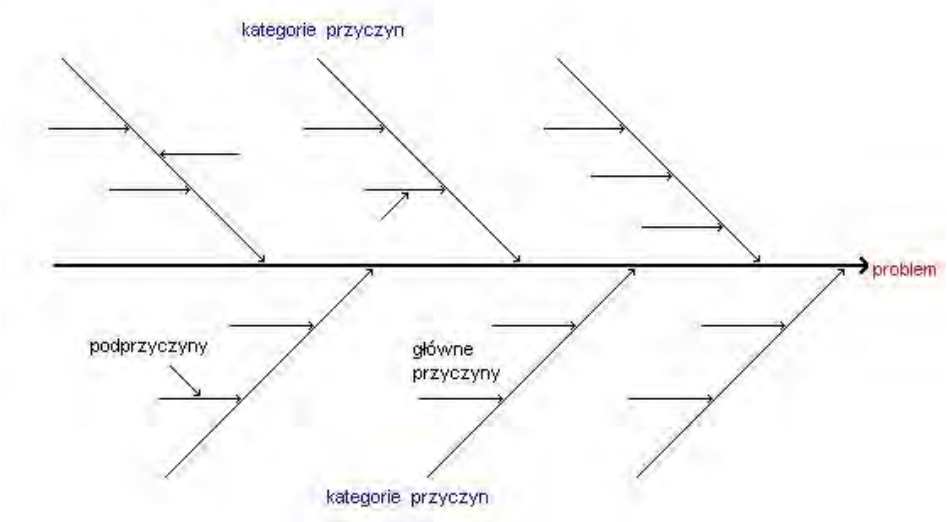


Rys. 3.3. Przykładowy diagram Pareto

Źródło: (Jayaswal, Patton, 2009)

Kolejną grupą diagramów są **diagramy przyczynowo-skutkowe**, tzw. diagramy Ikishawy lub „rybiej ości". Diagram taki służy do identyfikacji przyczyn istnienia określonego problemu. Diagram Ikishawy to diagramy drzewa błędów, który pomaga oddzielić przyczyny od skutków danej sytuacji i dzięki temu umożliwia dostrzeżenie złożoności problemu. Analiza rozpoczynana jest od stwierdzenia wystąpienia określonego skutku (np. błędu, awarii lub innego niepożądanego stanu) i prowadzona jest w kierunku identyfikacji wszystkich możliwych przyczyn, które go spowodowały. Na diagram składa się pięć historycznych głównych składowych - tzw. 5M: ludzie (ang. *Manpower)*, metody (ang. *Methods*), maszyny (ang. *Machinery*), materiały (ang. *Materials*), zarządzanie (ang. Management). Każdą z tych składowych rozbija się na poszczególne przyczyny, które powinny być rozpatrywane indywidualnie jako problemy do rozwiązania. Punktem wyjścia jest pozioma oś skierowana w prawą stronę, która jest określeniem sformułowanego problemu (skutku). Do niej dołączane są pochyłe strzałki które określają kategorie przyczyn. Przyczyny rozdziela się na podstawowe (główne) przyczyny oraz pod-przyczyny. Do każdej kategorii przyczyn przyporządkowane są poziome strzałki, które symbolizują główne przyczyny badanego problemu. Wykres rozbudowywany jest przez dołączanie kolejnych

przyczyn i pod-przyczyn (rys. 3.4).

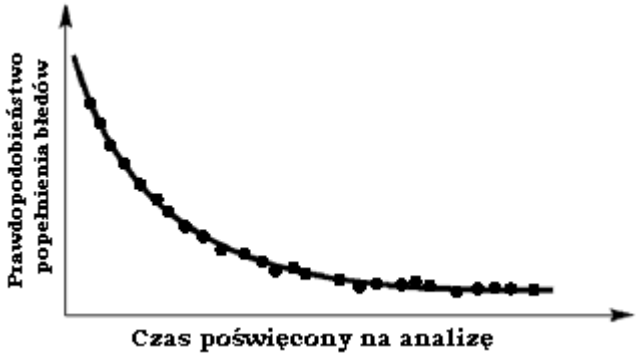


Rys. 3.4. Konstrukcja diagramu przyczynowo-skutkowego
Źródło: http://www.saferpak.com/cause_effect_articles/howto_cause_effect.pdf

Do typowo statystycznych narzędzi można zaliczyć **diagramy rozproszenia**. Pozwalają one na sprawdzenie istnienie związku (korelacji) pomiędzy dwoma zmiennymi. Odkrycie takiego związku oraz znajomość jego siły mogą być użyteczne w analizach przyczyn i skutków procesów. To ułatwia znalezienie sposobów rozwiązania danego problemu. Jednak trzeba wziąć pod uwagę, że korelacja może okazać się przypadkowa, dlatego diagramy rozproszenia nie mogą stanowić formalnego dowodu faktycznego istnienia związku. Dlatego korzystanie z diagramów rozproszenia zaleca się w celu potwierdzenia zależności, których istnienie się podejrzewa (np. wykryto je podczas burzy mózgów). Przykład diagramu rozproszenia wskazującego korelację ujemną przedstawia rys. 3.5. Wskazuje on, ze im więcej czasu i uwagi poświęconych zostanie projektowaniu, tym mniejsze jest prawdopodobieństwo wystąpienia poważnych błędów w projekcie.

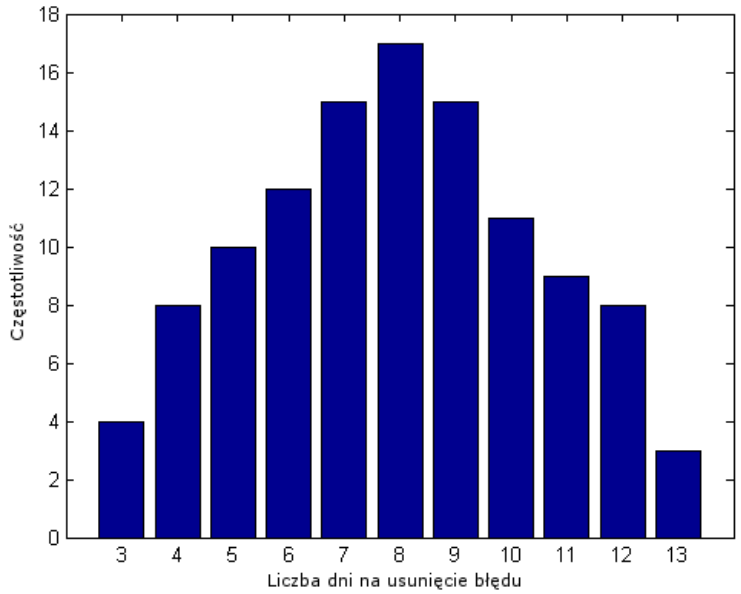
Kolejnym narzędziem są popularne **arkusze kontrolne**, czyli formularze zawierające listę elementów, które powinny zostać uzupełnione danymi. Taki arkusz pomaga w zbieraniu i analizie danych. Przykładowe zastosowania tej techniki to

analiza rodzajów i przyczyn błędów i pomiar zmienności określonego procesu.



Rys. 3.5. Przykład diagramu rozproszenia
Źródło: opracowanie własne na podstawie (Jayaswal, Patton, 2009)

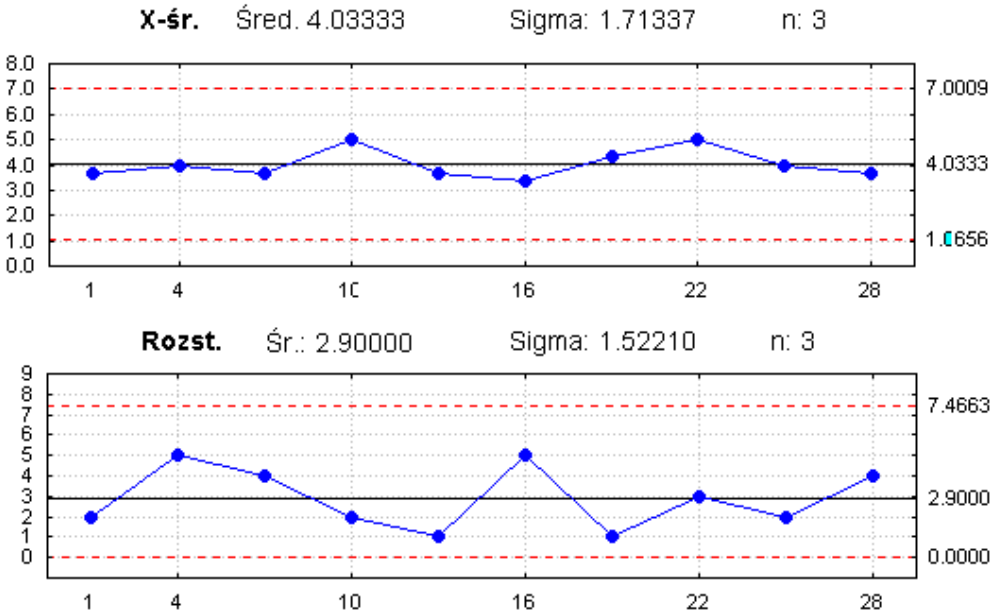
W celu wizualizacji danych arkusze kontrolne można łączyć z innymi technikami, np. **histogramami**. Histogram to wykres służący do analizy wariancji w zbiorach danych. Korzysta się niego, gdy konieczne jest poznanie rozproszenia danych np. ze względu na częstotliwość wystąpień zadanej wartości. Taka analiza pomaga określić rozkład danych procesu oraz umożliwia przewidywanie jego przebiegu w przyszłości. Przykład histogramu z rozkładem Gaussa prezentuje rys. 3.6. Przedstawia on rozkład liczby dni potrzebnych do usunięcia defektu w aplikacji.



Rys. 3.6. Przykładowy histogram
Źródło: opracowanie własne na podstawie (Jayaswal, Patton, 2009)

Karta kontrolna jest popularnym narzędzie wspomagające zarządzanie jakością i usprawnianiu procesów w organizacji. Założeniem, które narzuca się przy korzystaniu z kart kontrolnych jest zmienność procesu. Karta kontrolna służy do analizy tej zmienności, czyli wariancji parametru charakteryzującego proces. Jeśli wariancja ta okaże się zbyt duża, należy podjąć odpowiednie kroki. W ten sposób można uniknąć skutków niekontrolowanych procesów w projekcie czy organizacji. Parametrami najczęściej stosowanymi na kartach kontrolnych są wartości średnie, rozstępy (czyli różnice pomiędzy wartością największą a najmniejszą) i odchylenia standardowe.

Karta kontrolna powinna mieć dwie tzw. granice kontrolne definiujące akceptowalne limity odchylenia. Są to limity odpowiednio: górny (ang. U*pper Control Limit, UCL*) i dolny (ang. *Lower Control Limit, LCL*). Aby odczyt był prawidłowy, na karcie należy umieścić przynajmniej sto punktów danych. Badania wykazują (Florac, Carleton, 1999), że najbardziej przydatne są karty wartości średnich oraz rozstępów. Przykładową kartę kontrolną prezentuje rys. 3.7.



Rys. 3.7. Przykład karty kontrolnej
Źródło: (http://www.statsoft.pl)

Wszystkie przedstawione narzędzia mogą przyczynić się do wzrostu jakości oprogramowania. Ważne jest jednak, aby zbieranie i analizowanie danych było przeprowadzane skrupulatnie i systematycznie. Wybór odpowiednich narzędzi zależy od rodzaju i wielkości projektu. Narzędzia statystyczne przydadzą się przy dużych projektach, generujących sporą ilość danych do analiz. Diagramy, takie jak np. diagram przyczynowo-skutkowy, mogą być stosowane w każdym projekcie.

3.5. PYTANIA KONTROLNE

4. Podaj typowe przyczyny słabej jakości oprogramowania.
5. Wymień miary jakości oprogramowania.
6. Wymień znane Ci metody poprawy jakości.
7. Wymień i scharakteryzuj rodzaje testów oprogramowania.
8. Na czym polega konstrukcja diagramu przyczynowo-skutkowego?

4

Wzorce projektowe w metodykach zwinnych

Cel

Rozdział został poświęcony wzorcom projektowym stosowanym w metodykach zwinnych. Zawiera on genezę powstania oraz definicję wzorców projektowych. Przedstawiona została w nim klasyfikacja wzorców projektowych, a także wybrane ze względu na popularność wykorzystania wzorce. Wszystkie zaprezentowane wzorce zostały wzbogacone o przykłady zastosowania.

Plan

1. Projektowanie w metodykach zwinnych.
2. Definicja wzorca projektowego.
3. Klasyfikacja wzorców projektowych.
4. Opis oraz zastosowanie wybranych wzorców projektowych.

4.1. PROJEKTOWANIE W METODYKACH ZWINNYCH

W metodykach zwinny uwaga programistów skupiona jest na najbliższej iteracji, tak aby tworzone oprogramowanie jak najlepiej spełniało bieżące wymagania użytkowników. Zespoły nie poświęcają wiele uwagi przyszłym (tzn. nie dotyczącym bieżącej iteracji) wymaganiom i potrzebom użytkowników, gdyż mogą one ulec zmianie. Podejście zwinne do wytwarzania oprogramowania nie oznacza jednak całkowitej rezygnacji z architektury systemu oraz tworzenia jego projektu. Przeciwnie, prowadzi do stopniowej modyfikacji architektury tworzonego oprogramowania – jego ewolucji – tak aby zagwarantować elastyczność systemu i umożliwić wielokrotne wykorzystanie stworzonych elementów, komponentów.

Korzystanie z zasad projektowania obiektowego, w szczególności tych, które są wynikiem doświadczeń wielu pokoleń inżynierów oprogramowania, pozwala na eliminowanie negatywnego wpływu ciągłych zmian w projekcie, a także braku projektu na początku procesu wytwarzania oprogramowania (ang. *Big Design Up Front*). Zespoły pracujące w oparciu o metodyki zwinne stosują tylko te zasady, które mają istotny wpływ na projekt oprogramowania oraz sam kod źródłowy. Należą do nich przede wszystkim następujące zasady (Martin, 2008):

- zasada pojedynczej odpowiedzialności,
- zasada zamknięte-otwarte,
- zasada odwracania zależności,
- zasada podstawiania Liskov,
- oraz segregacji interfejsów.

Drugą ważną praktyką, ściśle powiązaną z zasadami projektowania obiektowego, jest stosowanie wzorców projektowych. **Wzorce projektowe** (ang. *Design Patterns*) ułatwiają rozwiązywanie często występujących problemów programistycznych i ochronę wybranych klas przed przyszłymi zmianami. Nie są one gotową implementacją rozwiązań, ale abstrakcyjnym opisem czy też szablonem zależności pomiędzy klasami i obiektami. Stosowane w projektach realizowanych metodykami zwinnymi pozwalają uniknąć problemów projektowych, które mogą się pojawić

w kolejnych iteracjach wraz z rozwojem projektu oraz oprogramowania.

W metodykach zwinnych, podobnie jak w tradycyjnych, do graficznej reprezentacji projektów systemów informatycznych wykorzystywany jest język modelowania **UML** (ang. *Unified Modelling Language*). Poszczególne wzorce w niniejszym rozdziale, będą przedstawione z wykorzystaniem wybranych notacji UML, tj. diagramów klas oraz sekwencji.

Wytwarzanie oprogramowania skierowane na zmiany ze względu na swój ewolucyjny charakter może prowadzić do negatywnych efektów w kodzie źródłowym. Symptomami negatywnych działań w projekcie są następujące elementy (Martin, 2011):

- **Sztywność** – wprowadzenie zmian w projekcie jest utrudnione, gdyż zmiana jednego elementu powoduje konieczność zmian elementów zależnych.
- **Wrażliwość** – zmiany w programie prowadzą do jego zepsucia.
- **Nieelastyczność** – nie jest zagwarantowana możliwość ponownego wykorzystania elementów oprogramowania, zasobów.
- **Niedostosowanie do rzeczywistości** – projekt utrudnia wprowadzanie zmian w sposób poprawny, a także sprzyja wolnym, nieefektywnym rozwiązaniom w kontekście środowiska działania.
- **Nadmierna złożoność**.
- **Niepotrzebne powtórzenia** – jako efekt nieumiejętnego stosowania narzędzi projektowych.
- **Nieprzejrzystość** – projekt jest niewłaściwie zorganizowany.

Stosowanie zasad programowania obiektowego pozwala na uniknięcie wymienionych powyżej symptomów, a tym samym na tworzenie oprogramowania wysokiej jakości.

Zasada pojedynczej odpowiedzialności (ang. *Single-Responsibility Principle - SRP*) jest jedną z najprostszych z przedstawionych reguł, ale także najtrudniejszą do stosowania w praktyce. Zasada ta głosi, że żadna klasa nie może być modyfikowana z więcej niż jednego powodu. Dlatego też pojedyncza klasa nie powinna odpowiadać za więcej niż jeden obszar działań (inaczej odpowiedzialności) jednocześnie, bo wtedy może istnieć więcej niż jeden powód jej modyfikowania.

W przypadku, gdy pojedyncza zmiana w systemie wymusza wprowadzenie całej sekwencji zmian w modułach zależnych, projekt nie jest elastycznym, a tym samym

można stwierdzić, że nie jest spełniona **zasada otwarte-zamknięte** (ang. *Open/Close Principle , OCP*). Komponenty oprogramowania, tj. moduły, klasy, funkcje, zbudowane zgodnie z zasadą otwarte-zamknięte, muszą być otwarte na rozszerzenia, ale także muszą być zamknięte do modyfikacji, co znaczy, że rozbudowa funkcjonalności danego elementu nie może skutkować zmianą już istniejącego kodu źródłowego.

Zasada podstawiania Liskov (ang. *Liskov Substitution Principle, LSP*) jest jednym z warunków zasady otwarte-zamknięte. Treść tej zasady mówi, że musi istnieć możliwość zastępowania typów bazowych ich podtypami. Daje możliwość rozbudowy klasy lub typu bazowego bez konieczności ich bezpośredniego modyfikowania, a tylko poprzez możliwość zastępowania podtypów.

Sytuacja, w której ogólna struktura jest uzależniona od szczegółowych rozwiązań w obszarze implementacji powoduje, że oprogramowanie staje się wrażliwe na zmianę szczegółów. **Zasada odwracania zależności** (ang. *Dependency Inversion Principle - DIP*) jest mechanizmem zapobiegającym tej wrażliwości, nakazującym moduły wysokopoziomowe uniezależniać od modułów niskopoziomowych. Przy czym zarówno moduły wysokopoziomowe, jak i niskopoziomowe, powinny być zależne od abstrakcji, a same abstrakcje nie powinny zależeć od szczegółowych rozwiązań.

Ostatnia z przytaczanych zasad to **zasada segregacji interfejsów** (ang. *Interface Segregation Principle, ISP*). Je celem jest usunięcie nadmiernie rozbudowanych i niespójnych interfejsów klas. Każdy rozbudowany interfejs powinien zostać podzielony na grupy mniejszych metod odpowiedzialnych za obsługę innego zbioru klientów.

Wzorce projektowe, jako rozwiązania „szablonowe”, stosują się do zasad projektowania obiektowego. Pozwalają mniej doświadczonym programistom uniknąć błędów związanych z niewłaściwym wykorzystaniem mechanizmów dziedziczenia, czy delegowania.

4.2. DEFINICJA WZORCA PROJEKTOWEGO

W różnych dziedzinach życia można rozpoznać pewne powtarzające się wzorce, rozwiązania, szablony. Niemniej samo pojęcie wzorca projektowego jako zestawu

pewnych sprawdzonych koncepcji pojawiło się po raz pierwszy w latach 80 XX wieku w architekturze. Jego twórcą jest amerykański architekt, Christopher Alexander, który uznał, że koncepcje architektoniczne można opisać za pomocą reguł definiujących gotowe rozwiązanie, które należy zastosować w celu osiągnięcia zamierzonego rezultatu, ale także określających kontekst jego wykorzystania.

Dokładna definicja wzorca, którą podał Christopher Alexander, brzmi (Alexander, 1977): „Wzorzec opisuje problem, który powtarza się wielokrotnie w danym środowisku, oraz podaje istotę jego rozwiązania w taki sposób, aby można było je zastosować miliony razy bez potrzeby powtarzania tej samej pracy". Jest ona na tyle ogólna, że możliwe było zastosowanie idei wzorca w innych dziedzinach, co też wpłynęło na jej upowszechnienie.

W informatyce formalne uznanie wzorców projektowych miało miejsce na początku lat 90, kiedy Erich Gamma opisał wzorce zastosowane w bibliotece ET++ implementującej GUI. Tekst Gamma wywołał dyskusje wśród programistów nad przedstawionymi rozwiązaniami, w efekcie czego powstała publikacja *Design Patterns - Elements of Reusable Object-Oriented Software* zawierająca usystematyzowany opis 23 popularnych wzorców. W publikacji dla każdego wzorca umieszczono opis, informacje dotyczące zastosowania oraz ograniczeń w użyciu, a także informacje o konsekwencjach oraz kompromisach wynikających z jego zastosowania (Cooper, 2001).

Wspomniana książka *Design Patterns - Elements of Reusable Object-Oriented Software* miała wielki wpływ na poszukiwanie i stosowanie wzorców projektowych. Jej autorzy (Gamma E., Helm R., Johnson R., Vlissides J.) określani są jako „**banda czworga**" (ang. *Gang of Four* – GoF).

Członkowie „bandy czworga" (GoF) przedstawili następująca definicję wzorca projektowego w dziedzinie inżynierii oprogramowania: „Wzorzec identyfikuje i specyfikuje pewną abstrakcję, której poziom znajduje się powyżej poziomu abstrakcji pojedynczej klasy, instancji lub komponentu" (Gamma, 1994). Definicja wzorca podkreśla fakt, że niektóre wzorce, prócz interakcji między obiektami, określają również strategię dziedziczenia oraz kompozycji tych obiektów. A wykorzystany w nich poziom abstrakcji zmusza programistę do odpowiedniego przystosowania „szablonu rozwiązania" do konkretnego problemu.

Każdy wzorzec projektowy definiowany jest w postaci czterech następujących elementów (Bruegge, 2011):

- **Nazwy** jednoznacznie identyfikującej go wśród pozostałych wzorców.
- **Opisu problemu**, w którym zastosowanie wzorca może być użyteczne.
- **Rozwiązania** w postaci zbioru współpracujących klas oraz interfejsów.
- **Zbioru konsekwencji** opisujących kompromisy i alternatywne rozwiązania dla problemów, w których danych wzorzec może być użyteczny.

4.3. KLASYFIKACJA WZORCÓW PROJEKTOWYCH

Wzorce projektowe mogą dotyczyć wielu poziomów abstrakcji projektowanych systemów – od bardzo specyficznych rozwiązań po bardzo ogólne. W literaturze można znaleźć setki wzorców i wciąż odkrywane są nowe związane nie tylko z problemami z zakresu projektowania oprogramowania, ale także modelowania procesów, zarządzania zależnościami, zarządzania konfiguracją, adaptacją metodologii, zarządzania zasobami, itd.

W książce *Design Patterns - Elements of Reusable Object-Oriented Software,* tak jak wspomniano, opisano 23 wzorce projektowe na średnim poziomie uogólnienia. Autorzy (GoF) podzielili przedstawione wzorce na trzy typy ze względu na ich zastosowanie:

- **Wzorce konstrukcyjne** (ang. *Creational Patterns*). Wykorzystuje się je do pozyskiwania obiektów zamiast bezpośrednio tworzenia instancji klasy. W konsekwencji, dzięki możliwości decydowania jakiego typu obiekt ma zostać utworzony w danym przypadku, oprogramowanie zyskuje na elastyczności. Wzorce konstrukcyjne wraz ich krótką charakterystyką zostały przedstawione w tab. 4.1.
- **Wzorce strukturalne** (ang. *Structural Patterns*). Pomagają łączyć obiekty w większe struktury. Ich lista została przedstawiona w tab. 4.2.
- **Wzorce czynnościowe** lub behawioralne (ang. *Behavioral Patterns*). Pomagają zdefiniować komunikację pomiędzy obiektami oraz kontrolować przepływ danych. Wzorce czynnościowe wraz z opisem zawiera tab. 4.3.

Tabela 4.1. Wzorce konstrukcyjne GoF

Budowniczy (ang. *Builder*)	Oddziela konstruowanie złożonych obiektów od ich reprezentacji w celu umożliwienia tworzenia różnych reprezentacji obiektów za pomocą tych samych zabiegów konstrukcyjnych.
Fabryka abstrakcji (ang. *Abstract Factory*)	Dostarcza interfejs umożliwiający tworzenie rodzin spokrewnionych lub uzależnionych od siebie obiektów bez konieczności specyfikowania ich konkretnych klas.
Metoda fabrykująca (ang. *Factory Method*)	Prosta klasa decyzyjna, która w zależności od otrzymanych danych, zwraca obiekt jednej z wielu klas pochodnych abstrakcyjnej klasy podstawowej.
Prototyp (ang. *Prototype*)	Pozwala na klonowanie istniejącej instancji klasy. Właściwości instancji mogą być następnie modyfikowane za pomocą metod publicznych.
Singleton	Ogranicza możliwość tworzenia obiektów danej klasy do jednej instancji udostępnianej globalnie.

Tabela 4.2. Wzorce strukturalne GoF

Adapter	Używany do stworzenia nowego interfejsu dla klasy, tak aby dostosować go do interfejsu innej klasy.
Dekorator (ang. *Decorator*)	Pozwala na dynamiczne dodawanie nowych cech do obiektu.
Fasada (ang. Facade)	Używany do utworzenia pojedynczej klasy reprezentującej złożony system, która ułatwi jego użycie.
Kompozyt (ang. *Composite*)	Pozwala na takie zgrupowanie różnych obiektów, aby mogły być one traktowane przez klienta w jednakowy sposób.
Most (ang. *Bridge*)	Oddziela interfejs obiektu od jego implementacji, tak aby mogły być niezależnie modyfikowane.
Pełnomocnik (ang. *Proxy*)	Tworzy prosty obiekt zastępujący obiekt bardziej złożony, który może być dostępny w późniejszym czasie.
Waga piórkowa (ang. *Flyweight*)	Wzorzec stosowany do obiektów współdzielonych. Instancja nie zawiera informacji o stanie, lecz stan jest przechowywany na zewnątrz.

Tabela 4.3. Wzorce czynnościowe GoF

Interpreter	Definiuje, w jaki sposób wprowadzić do programu składowe języka na podstawie jego gramatyki i opisów semantycznych, aby możliwe stworzenie interpretera.
Iterator	Formalizuje sposób poruszania się po kolekcji danych o dowolnej strukturze.
Łańcuch odpowiedzialności (ang. *Chain of Responsibility*)	Uwalnia nadawcę żądania od konieczności powiązania żądania z jednym odbiorcą. Obiekty łańcucha odpowiedzialności przekazują kolejno między sobą żądania do czasu aż zostanie ono rozpoznane i obsłużone przez jeden z obiektów.
Mediator	Definiuje sposób uproszczenia komunikacji pomiędzy obiektami z użyciem osobnego obiektu, zapobiegając jawnym odwołaniom między nimi.
Metoda szablonowa (ang. *Template*)	Dostarcza abstrakcyjną definicję algorytmu.
Momento	Celem wzorca jest udostępnienie stanu wewnętrznego obiektu innym obiektom w taki sposób, aby nie naruszyć jego hermetyzacji. Działanie takie umożliwia zapamiętanie, przechowywanie oraz odtworzenie stanu obiektu.
Obserwator (ang. *Observer*)	Definiuje sposób powiadamiania o zmianach stanu obiektu inne obiekty.
Polecenie (ang. *Command*)	Nadaje żądaniu formę obiektu, pozwala na zrealizowanie dziennika poleceń i wycofywanie wykonanych operacji.
Stan (ang. *State*)	Wzorzec umożliwia obiektowi zmianę jego zachowania w wyniku zmiany jego stanu wewnętrznego.
Strategia (ang. *Strategy*)	Definiuję rodzinę obudowanych algorytmów i umożliwia wymienne stosowanie w trakcie działania programu.
Wizytator (ang. *Visitor*)	Umożliwia zmianę operacji wykonywanych na elementach klasy bez zmiany struktury tej klasy.

Prócz podziału zaproponowanego przez GoF, istnieją również inne klasyfikację wzorców projektowych, bądź też rozszerzenia podstawowej klasyfikacji, na przykład o wzorce współbieżności, czy mapowania obiektowo-relacyjnego.

Do interesujących wzorców wykorzystywanych w metodykach zwinnych należą również wzorce architektury związane z graficznym interfejsem użytkownika, jak

Model-Widok-Kontroler (ang. *Model-View-Controller, MVC*) lub Model-Widok-Prezenter (ang. *Model–View–Presenter, MVP*), oraz wzorce zarządzania zasobami, jak *Wzorzec chciwego pozyskiwania zasobów* (ang. *Eager Acquisition Pattern*), *Wzorzec leniwego pozyskiwania zasobów* (ang. *Lazy Acquisiton Pattern*), *Wzorzec ponownego wykorzystania zasobów* (ang. *Caching*), czy też *Wzorzec wyszukiwania* (ang. *Lookup*).

Poniżej zostały przedstawione również inne klasyfikacje wzorców projektowych:

- Warstwy w aplikacjach biznesowych,
- Wzorce logiki aplikacji,
- Wzorce infrastruktury komponentowej,
- Wzorce architektury źródła danych,
- Wzorce mapowania obiektowo-relacyjnego,
- Wzorce prezentacji,
- Wzorce dystrybucji,
- Wzorce stanu sesji,
- Wzorce zasobowe,
- Wzorce przetwarzania równoległego,
- Wzorce komunikacji zdalnej,
- Wzorce podstawowe.

4.4. WYBRANE WZORCE PROJEKTOWE GOF

Wzorce projektowe GoF to wzorce ogólnego przeznaczenia. Umiejętność posługiwanie się tymi wzorcami jest niezbędną kompetencją programisty i bazą, dzięki której może on efektywnie korzystać z złożonych wzorców oraz zaawansowanych technologii. W dalszej części podrozdziału omówiono następujące wzorce: *Metoda fabrykująca, Fabryka abstrakcyjna, Adapter, Most, Kompozyt, Fasada, Polecenie, Obserwator, Pełnomocnik* i *Strategia*.

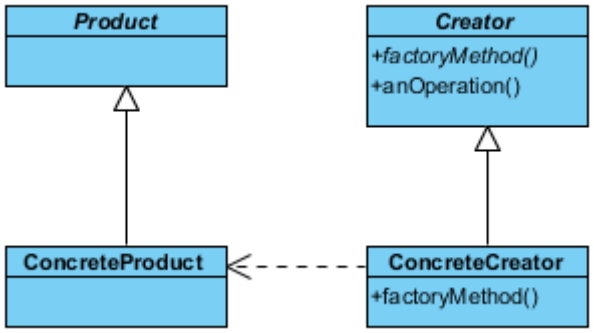
Metoda fabrykująca (ang. *Factory Method*)

Wzorzec *Metoda fabrykująca* jest podstawowym wzorcem kreacyjnym. Jego celem

jest zdefiniowanie dedykowanego interfejsu do tworzenia obiektów. Interfejs ten ma umożliwić hermetyzację sposobu wyboru klasy tworzonego obiektu spośród jej klas pochodnych oraz użytego konstruktora. Wzorzec także umożliwia przekazanie odpowiedzialność za tworzenie obiektów do klas pochodnych.

Zgodnie z diagramem klas przestawionym na rys. 4.1. *Metoda fabrykująca* złożona jest z dwóch klas abstrakcyjnych (bądź interfejsów) – klasy `Creator`, która zawiera metodę wytwórczą o nazwie `factoryMethod()` tworzącą produkty, oraz klasy `Product` reprezentującej wszystkie tworzone produkty.

Obie klasy abstrakcyjne posiadają implementacje w postaci klas konkretnych, tj. `ConcreteCreator` dla klasy `Creator` oraz `ConcreteProduct` dla klasy `Product`. W klasie `ConcreteCreator` zostaje przeciążona metoda wytwórcza `factoryMethod()`, tak aby klasa tworzyła obiekty klasy `ConcreteProduct`. Dzięki temu rozwiązaniu podczas zmiany obiektu `Creator` jednocześnie zmieniany jest tworzony produkt.

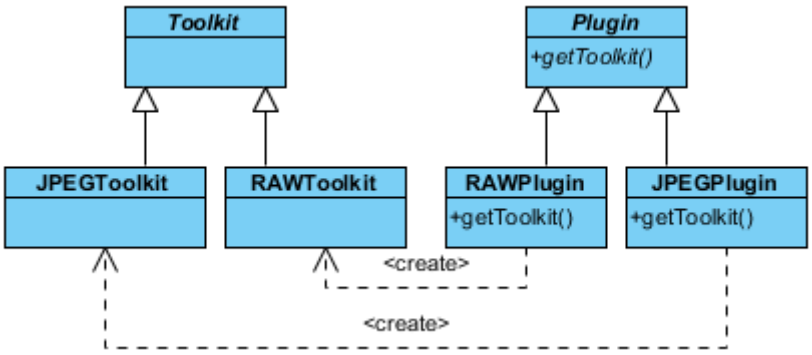


Rys. 4.1. Diagram klas wzorca Metoda fabrykująca
Źródło: Opracowanie własne na podstawie (Freeman, 2004)

Z punktu widzenia klienta metoda wytwórcza jest równoważna z konstruktorem, tzn. jej wywołanie powoduje utworzenie obiektu określonego typu. Przy czym bezpośrednie wywołanie konstruktora przez klienta zawsze powoduje utworzenie obiektu konkretnej klasy, natomiast użycie wzorca *Factory Method* pozwala metodzie tworzącej obiekty na wybór klasy obiektu i sposobu jego tworzenia.

W podstawowej postaci wzorca użycie odpowiedniej klasy `ConcreteCreator` determinuje klasę i właściwości produktu (`ConcreteProduct`), jaki zostanie utworzony. W innej wersji tego wzorca `Creator` jest klasą, której statyczna metoda `factoryMethod()` dokonuje selekcji produktów na podstawie przekazanych jej parametrów.

Przykładem wykorzystania tego wzorca może być przeglądarka plików graficznych, która wykorzystuje system wtyczek do obsługi poszczególnych formatów plików (rys. 4.2). Metoda wytwórcza `getToolkit()` klasy dziedziczącej po klasie `Plugin` zwraca wskaźnik do obiektu klasy, która może manipulować obrazami danego formatu. Dzięki takiemu rozwiązaniu możliwe jest rozszerzanie listy obsługiwanych formatów.



Rys. 4.2. Zastosowanie wzorca Metoda fabrykująca w przeglądarce plików graficznych
Źródło: Opracowanie własne

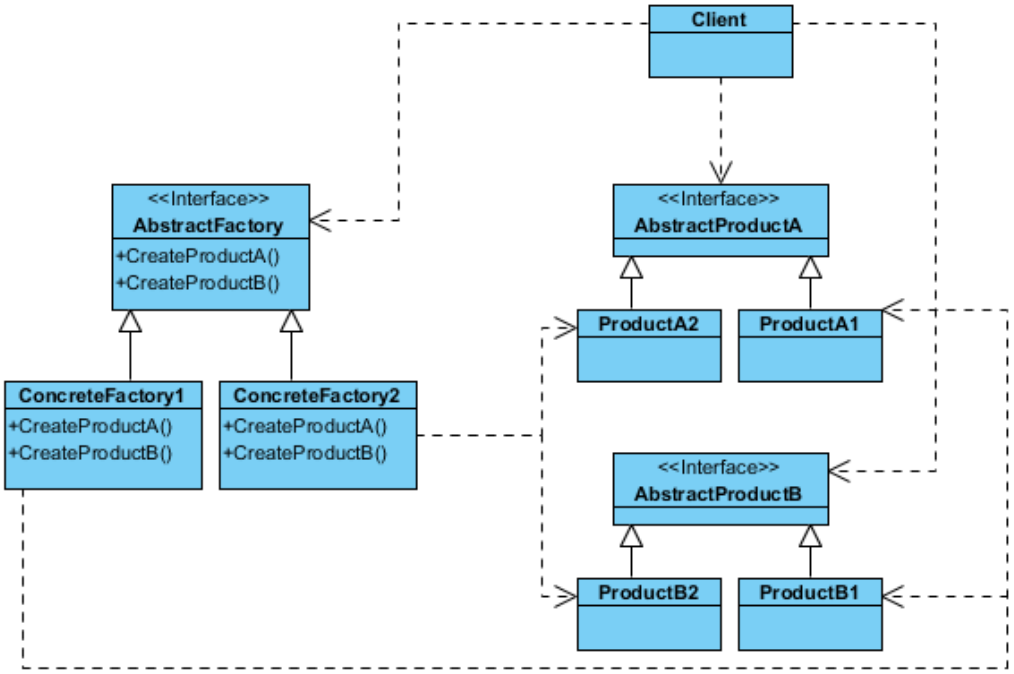
Użycie wzorca *Metoda fabrykująca* należy rozważyć w przypadku, gdy klasa używa swoich klas pochodnych w celu określenia, jaki rodzaj obiektu ma być utworzony. A także w przypadku, gdy informacje o tym, jaki rodzaj obiektu ma zostać utworzony, mają być ulokowane w klasach pochodnych.

Fabryka abstrakcyjna (ang. *Abstract Factory*)

Wzorzec *Fabryka abstrakcyjna* jest rozwinięciem koncepcji wzorca *Metoda fabrykująca*. Wzorzec jest wykorzystywany do tworzenia jednej z wielu związanych ze

sobą klas obiektów, z których każdy może na żądanie zwrócić wiele innych obiektów. Udostępniany przez wzorzec interfejs służy odseparowaniu klienta od różnych platform dostarczających różne implementację tego samego zbiory obiektów.

Klasa `AbstractFactory` (rys. 4.3), analogicznie do rozwiązania stosowanego we wzorcu *Metoda fabrykująca*, odpowiedzialna jest za tworzenie poszczególnych grup produktów. `AbstractFactory` jest klasą abstrakcyjną, a więc nie definiuje, w jaki sposób mają być tworzone odpowiednie produkty; deklaruje jedynie obecność odpowiedzialnych za to metod. Tworzeniem konkretnych produktów (np. `ProductA1`, `ProductB1`) zajmują się jej implementacje w odpowiednich klasach konkretnych (`ConcreteFactory1`).



Rys. 4.3. Diagram klas wzorca Fabryka abstrakcyjna
Źródło: Opracowanie własne na podstawie (Freeman, 2004)

Klient, wybierając odpowiednią fabrykę `ConcreteFactory`, decyduje jednocześnie o wyborze całej rodziny produktów. Pozwala to w łatwy sposób zmieniać

całe rodziny produktów zmieniając tylko implementacje fabryki. A tym samym zapewnić większą elastyczność systemu, gdyż implementacje zarówno klas fabryki, jak i rodziny produktów, są niewidoczne dla klienta.

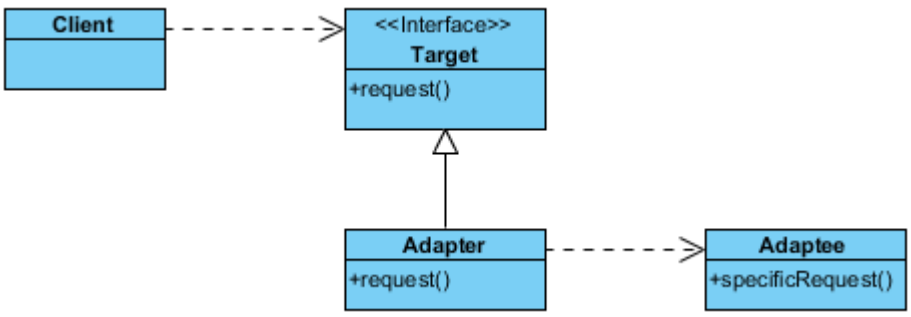
Dodanie nowej rodziny produktów wymaga implementacji nowej fabryki, która będzie dostarczała te produktu, dlatego wzorzec ten stosuje się w przypadkach, gdy zestaw produktów jest zamknięty.

Jedną z implementacji wzorca *Fabryka abstrakcyjna* jest aplikacja z obsługą wielu odmian interfejsu użytkownika, jak Windows i Macintosh. Informując fabrykę, że GUI aplikacji ma mieć wygląd środowiska Windows, zwraca ona implementację dostarczającą obiekty (okna, pola wybory, typy przycisków) przeznaczone do tego środowiska.

Adapter

Wzorzec *Adapter* konwertuje interfejs przestarzałej klasy na inny interfejs oczekiwany przez obecnego klienta, dzięki czemu klient może współdziałać z przestarzałą klasą.

We wzorcu występują trzy podstawowe klasy – `Target`, `Adapter`, `Adaptee` (rys. 4.4). Klasa `Target` jest interfejsem wymaganym przez klienta. Obiektem dostarczającym żądaną przez klienta funkcjonalność, ale o interfejsie niezgodnym z oczekiwaniami klienta, jest klasa `Adaptee`. Natomiast klasa `Adapter` deleguje żądania klienta do przestarzałej klasy `Adaptee`, wykonując przy okazji niezbędne konwersje, a więc zapewnia ona komunikację pomiędzy klasą `Target` a `Adaptee`.



Rys. 4.4. Diagram klas wzorca Adapter

Źródło: Opracowanie własne na podstawie (Freeman, 2004)

Dzięki zastosowaniu wzorca *Adapter*, klient może współpracować z klasą `Adaptee` bez konieczności modyfikowania któregokolwiek z interfejsów. Co więcej, `Adapter` może dodawać funkcjonalność do klasy `Adaptee` oraz współpracować z dowolną klasą pochodną przestarzałej klasy `Adaptee`. Niemniej dla każdej nowej specjalizacji interfejsu klienckiego (`Target`) wymagany jest nowa klasa `Adapter`.

Prócz zaprezentowanej na rys. 4.4 realizacji wzorca poprzez dziedziczenie, istnieje możliwość jego implementacji poprzez kompozycję. W tym przypadku należy zawrzeć klasę `Adaptee` wewnątrz klasy `Adapter` oraz utworzyć metody wymaganego interfejsu realizujące wywołania metody klasy wewnętrznej. Aby zastosować to rozwiązanie, język programowania musi umożliwiać stosowanie wielokrotnego dziedziczenia lub dziedziczenia i implementacji interfejsu.

Wzorzec *Adapter* jest przydatny szczególnie w przypadkach aplikacji, w których planowane jest wykorzystanie gotowych bibliotek, które udostępniają interfejsy niezgodne z interfejsami zastosowanymi w aplikacji.

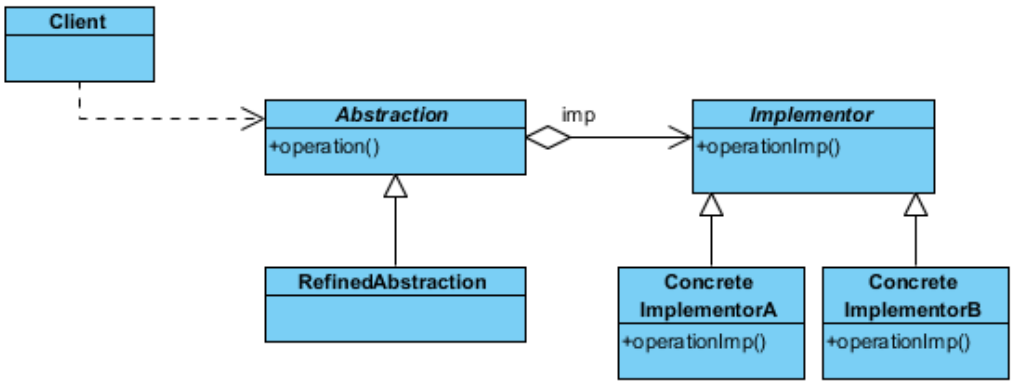
Most (ang. *Bridge*)

Wzorzec *Most* może się wydawać podobny do wzorca *Adapter*, bo również konwertuje jeden rodzaj interfejsu na inny. Jednak intencje opracowania wzorców były rożne. Wzorzec *Most* został zaprojektowany, aby odseparować interfejs klasy od jego implementacji. Dzięki czemu możliwa jest zmiana jego implementacji w dowolnym czasie bez wprowadzania zmian w kodzie, który korzysta z klasy.

Na rys. 4.5 został przedstawiony diagram klas wzorca *Most*. W klasie `Abstraction` definiowany jest interfejs widoczny dla klienta. Natomiast `Implementor` jest abstrakcyjną klasą definiującą metody niższego poziomu dostępne dla klasy `Abstraction`. Klasa `Abstraction` utrzymuje referencje do powiązanego z nią obiektu klasy pochodnej `Implementor`, tj. `ConcreteImplementorA` lub `ConcreteImplementorB`.

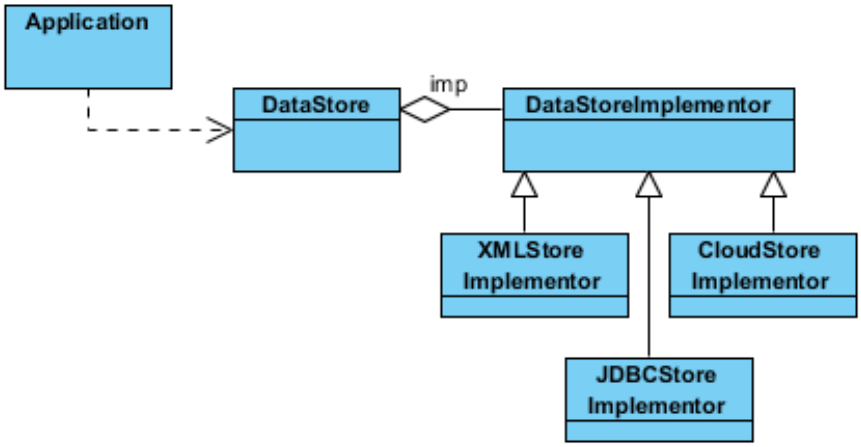
Dzięki zastosowaniu powyższego rozwiązania, klient zostaje oddzielony od abstrakcji i konkretnej implementacji, a więc interfejsy i implementacje mogą być rozwijane niezależnie. Wzorzec pozwala zachować stały interfejs klienta podczas

dokonywania zmian w klasach implementujących. W przypadku kompilacji, nie jest konieczne rekompilowanie modułów klienta, a jedynie samego mostu i klas implementujących.



Rys. 4.5. Diagram klas wzorca Most

Źródło: Opracowanie własne na podstawie (Gamma, 1994)



Rys. 4.6. Zastosowanie wzorca Most do testowania mechanizmów przechowywania danych

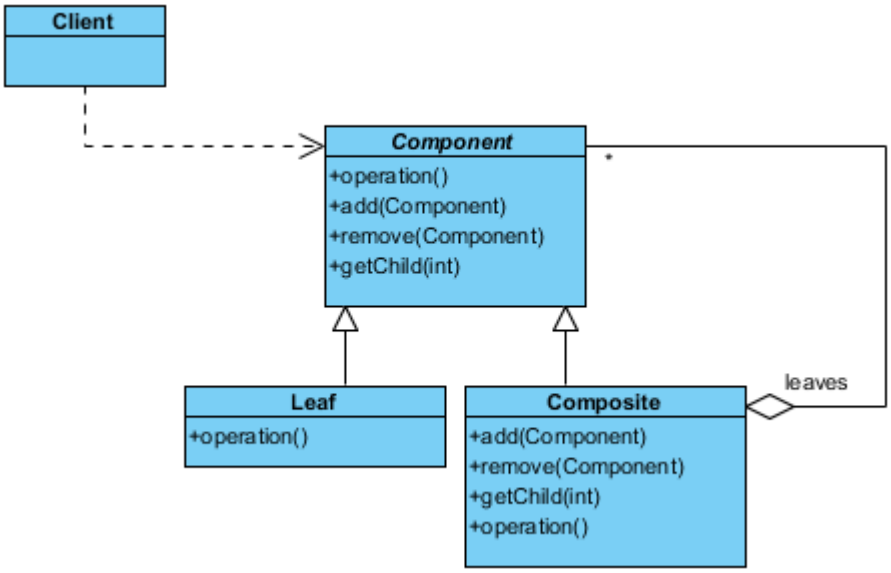
Źródło: Opracowanie własne

Wzorzec *Most* jest użyteczny podczas wykonywania testów różnych implementacji tego samego interfejsów oraz w przypadku aplikacji graficznych i okienkowych, które muszą pracować na różnych platformach. Zastosowanie wzorca do testowania trzech

różnych mechanizmów przechowywania danych w aplikacji zostało zobrazowane na rys. 4.6.

Kompozyt (ang. *Composite*)

Wzorzec *Kompozyt* pozwala na tworzenie hierarchii o zmiennej głębokości i szerokości, dzięki czemu liście i węzły pośrednie w drzewie hierarchii mogą być obsługiwane w jednolity sposób. *Kompozyt* jest kolekcją obiektów, z których każdy może być kompozytem lub pojedynczym obiektem.



Rys. 4.7. Diagram klas wzorca Kompozyt

Źródło: Opracowanie własne na podstawie (Gamma, 1994)

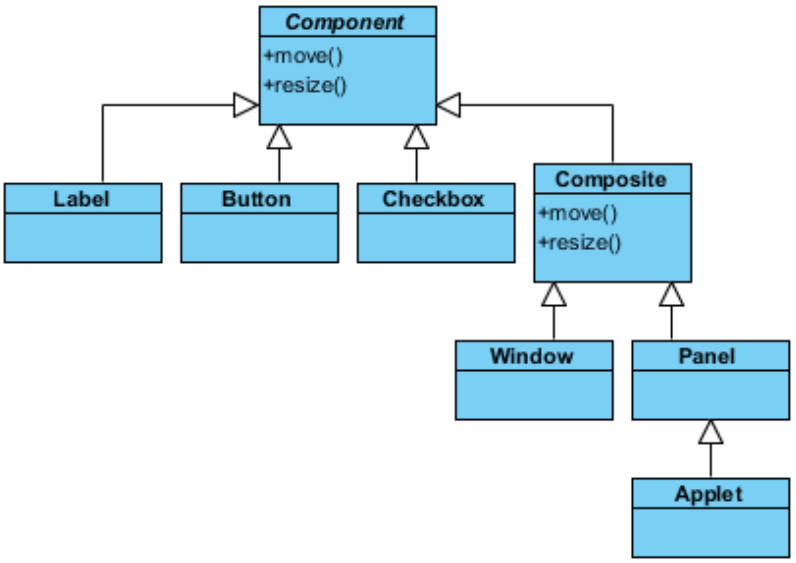
Głównym elementem wzorca jest klasa abstrakcyjna (czy też interfejs) `Component` reprezentująca dowolny element w hierarchii drzewa (rys. 4.7). Klasa `Component` posiada metody umożliwiające dodanie i usuwanie obiektów potomnych, odwoływanie się do dowolnego obiektu potomnego (`getChild()`) oraz wykonanie na nim metody `operation()`.

Klasa `Component` posiada dwie implementacje. Pierwszą jest klasa `Leaf` (pol. *liść*), która nie posiada elementów potomnych. Drugą jest klasa `Composite`

reprezentująca dowolny element pośredni – w nomenklaturze drzew nazywany węzłem – który może posiadać wiele potomków oraz nimi zarządzać, tzn. dodawać i usuwać obiekty potomne. Klasa `Composite` może wykonywać także specyficzną operację `operation()` dla dowolnego węzła, który następnie deleguje wywołanie operacji do obiektów potomnych.

Z punktu widzenia klienta, możliwe jest zarządzanie całą strukturą drzewa za pomocą jednego obiektu tzn. korzenia drzewa.

Przykładem zastosowania wzorca jest aplikacja, która wykorzystuje widżety jako elementy interfejsu graficznego (rys. 4.8). Elementy te mogą być organizowane w grupy, obsługiwane (np. przesuwane) w sposób zintegrowany. Grupowanie może mieć charakter rekurencyjny (Bruegge, 2011).



Rys. 4.8. Zastosowanie wzorca Kompozyt do organizacji elementów GUI w aplikacji
Źródło: Opracowanie własne na podstawie (Bruegge, 2011)

Wzorzec *Kompozyt* upraszcza znacznie program kliencki, dając możliwość reprezentacji prostych i złożonych elementów w strukturze hierarchii z ujednoliconym interfejsem. Z drugiej strony, chociaż wzorzec pozwala w łatwy sposób dodawać nowe rodzaje obiektów, które dostarczają podobny interfejs, wymusza zbytnie uogólnianie

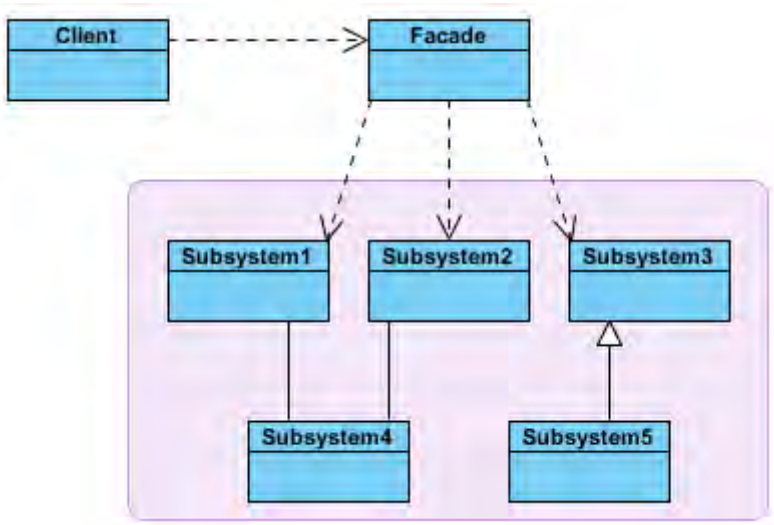
interfejsów poszczególnych klas, przez co ich interfejsy mogą być nieczytelne lub nieoptymalne.

Fasada (ang. *Facade*)

Wzorzec *Fasada* jest prostym wzorcem strukturalnym. Używany jest do obudowania zbioru złożonych klas (czy też podsystemów) w celu uproszczenia interfejsu dla klienta. Otrzymane uproszczenie może zmniejszać elastyczność obudowywanych klas, ale często dostarcza wszystkie niezbędnych funkcje. Natomiast obudowywane klasy i ich metody są nadal dostępne dla bardziej zaawansowanego klienta.

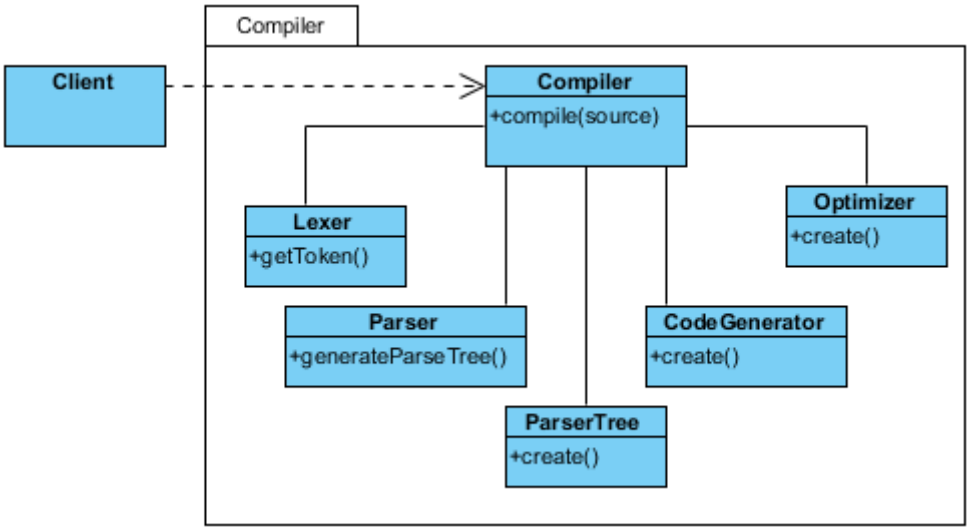
We wzorcu (rys. 4.9), pojedyncza klasa `Facade` implementuje wysokopoziomowy interfejs podsystemu poprzez wywoływanie metod klas na niższym poziomie. Stanowi więc ona dodatkowa warstwę abstrakcji w dostępie do klas oraz podsystemów.

Klasa `Facade` musi znać ich strukturę i przeznaczenie poszczególnych klas/podsystemów. Żądania, przesyłane przez klienta klasie `Facade`, są przez nią delegowane do odpowiednich klas/podsystemów. Dlatego też problem konfiguracji i obsługi złożonych podsystemów zostaje przerzucony z klienta na klasę fasady, w konsekwencji czego osiągana jest hermetyzacja podsystemów.



Rys. 4.9. Diagram klas wzorca Facade
Źródło: Opracowanie własne na podstawie (Freeman, 2004)

Przykładem wykorzystania wzorca *Fasada* może być podsystem kompilatora złożony z analizatora leksykalnego, parsera, drzewa rozbioru syntaktycznego, generatora kodu wynikowego i optymalizatora (rys. 4.10). Użytkownik kompilujący kod źródłowy uzyskuje dostęp wyłącznie do klasy `Compiler`, wywołującej odpowiednie metody pozostałych klas, aby zachować hermetyzację podsystemu (Bruegge, 2011).



Rys. 4.10. Zastosowanie wzorca Fasada dla podsystemu kompilatora
Źródło: Opracowanie własne na podstawie (Bruegge, 2011)

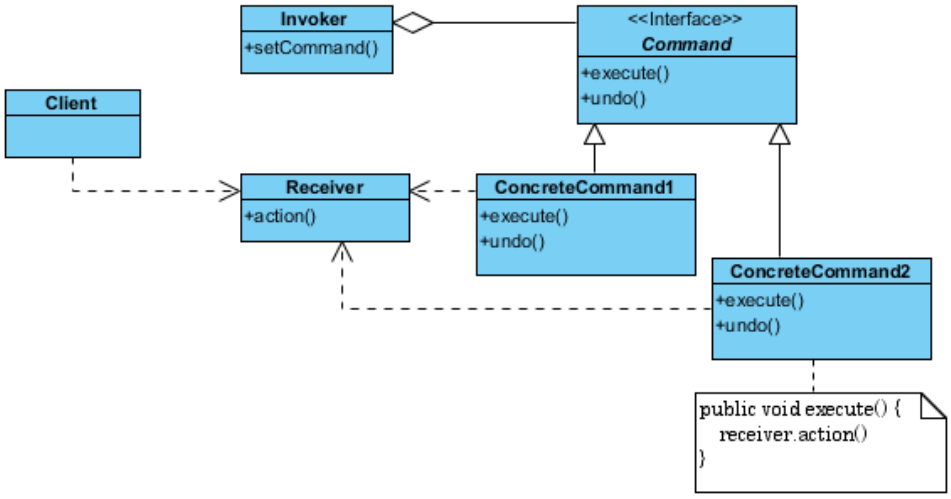
Polecenie (ang. *Command*)

Wzorzec *Polecenie* jest przykładem wzorca czynnościowego. Umożliwia hermetyzację żądań w celu ich wykonania, wycofania i kolejkowania w ujednolicony sposób.

Wzorzec złożony jest z następujących klas: `Command`, `ConcreteCommand`, `Invoker` oraz `Receiver` (rys. 4.11). Podstawowym elementem jest interfejs *Command* deklarujący metodę wykonania polecenia `execute()` oraz metodę wycofania polecenia `undo()`. Metody te są implementowane w klasach pochodnych – w tym wypadku `ConcreteCommand1` oraz `ConcreteCommand2` – w postaci polecenia wykonania określonej akcji (`action()`) na obi ekcie klasy odbiorcy

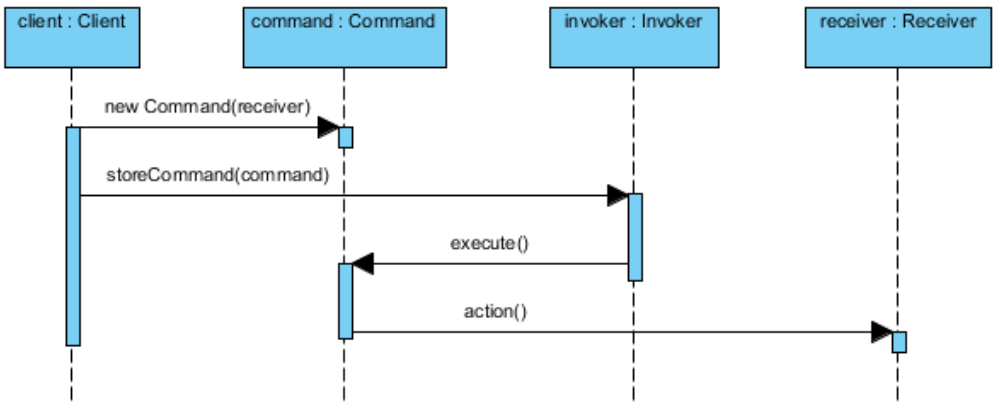
`Receiver`. Każda z klas `ConcreteCommand` hermetyzuje usługi przeznaczone dla konkretnego odbiorcy (`Receiver`). Z kolei klasa `Invoker` jest odpowiedzialna za wykonywanie i wycofywanie poleceń.

Szczegółowy przepływ sterowania we wzorcu przedstawia diagram sekwencji (rys. 4.12).



Rys. 4.11. Diagram klas wzorca Polecenie

Źródło: Opracowanie własne na podstawie (Bruegge, 2011)



Rys. 4.12. Diagram sekwencji wzorca Command

Źródło: Opracowanie własne na podstawie (Gamma, 1994)

Przykładem zastosowania wzorca *Polecenie* może być stos akcji użytkownika do wycofania dla aplikacji edytora graficznego. Wszystkie dostępne dla użytkownika polecenia powinny być obiektami klas pochodnych abstrakcyjnej klasy `Command` i implementować operacje wykonania, wycofania oraz przywrócenia polecenia. Operacje wykonane umieszczane powinny być na stosie akcji, a w przypadku wycofania akcji do obiektu polecenia na stosie wysyłany jest komunikat skutkujący wykonaniem metody wycofania (Bruegge, 2011).

Dzięki zastosowaniu wzorca *Polecenie,* `Invoker` oddzielony jest od konkretnego polecenia, a obiekt odbiorcy (`Receiver`) oddzielony jest od algorytmu polecenia (`ConcreteCommand`).

Obiekty konkretnych poleceń (`ConcreteCommand`) mogą być tworzone, niszczone i magazynowane, a także nowe polecenia mogą być dodawane bez zmian w istniejącym kodzie.

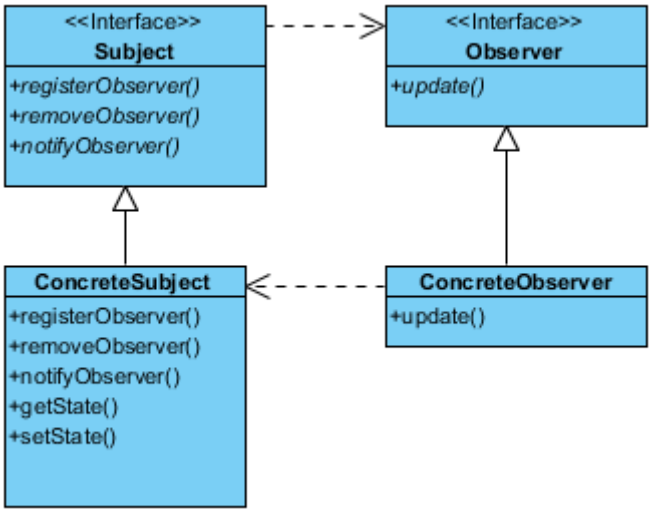
Obserwator (ang. *Observer*)

Celem wzorca *Observer* jest utworzenie zależności typu jeden – wiele pomiędzy obiektami oraz umożliwienie intensywnego przepływu komunikatów o zmianie stanu obiektu wyróżnionego do pozostałych obiektów zwanych obserwatorami lub subskrybentami.

W skład wzorca wchodzi obiekt obserwowany (`Subject`), którego głównym zadaniem jest utrzymanie informacji o stanie, oraz jeden lub więcej obiektów będących obserwatorami (`Observer`) wykorzystujących tę informację (rys. 4.13).

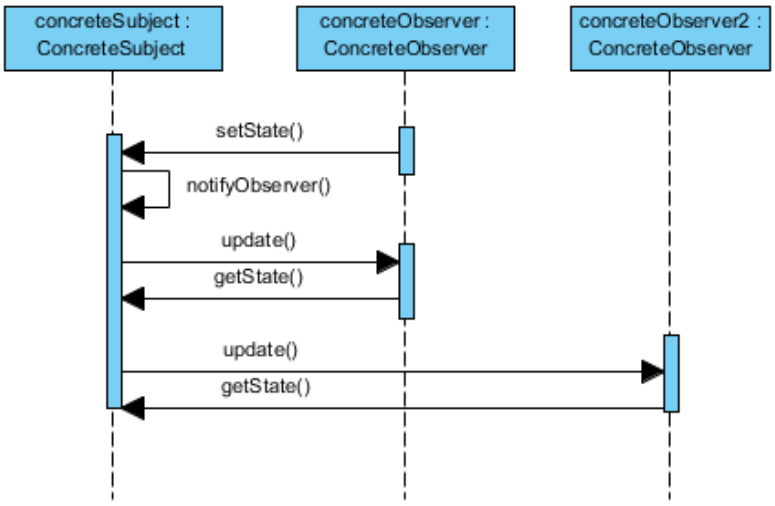
Obiekt `Subject` posiada metody pozwalające obserwatorom na zarejestrowanie zapotrzebowania na powiadomienie o każdej zmianie jego stanu (`registerObserver()`) oraz usuwania zapotrzebowania (`removeObserver()`).

Każdy konkretny obserwator (`ConcreteObserver`) definiuje także metodę `update()`, stanowiącą realizację wspomnianego powiadomienia. Tzn. w przypadku, gdy obiekt `Subject` zmienia swój stan, wywołuje swą metodę `notify()`, która powoduje wywołanie metody `update()` każdego zarejestrowanego obserwatora. Mechanizm ten został przedstawiony na diagramie sekwencji na rys. 4.14.



Rys. 4.13. Diagram klas wzorca Obserwator

Źródło: Opracowanie własne na podstawie (Freeman, 2004)



Rys. 4.14. Diagram sekwencji wzorca Obserwator

Źródło: Opracowanie własne na podstawie (Gamma, 1994)

Wzorzec *Obserwator* wprowadza abstrakcyjne połączenie z obiektem obserwowanym – `Subject`. Pozwala zachować spójność pomiędzy warstwami

aplikacji, ponieważ informacje o zmianach w jednej warstwie są przekazywane natychmiast do pozostałych obiektów. Jest to wykorzystywane do komunikacji w wielu aplikacjach okienkowych. Zamiennie zamiast nazwy `Observer` wykorzystuje się także nazwę `Listener`.

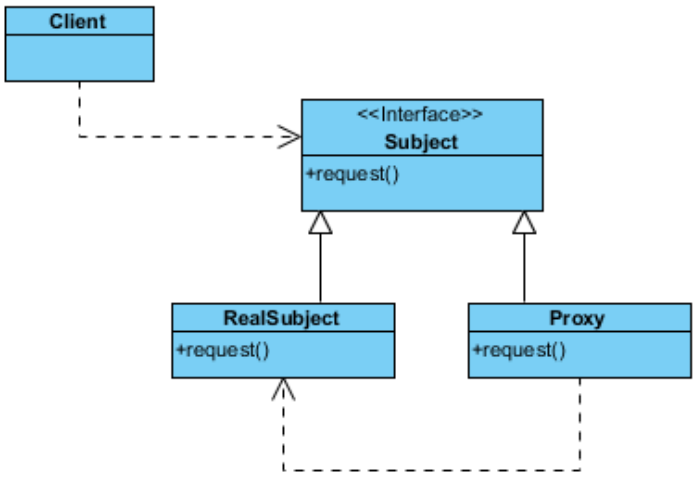
Powtarzające się uaktualnienia obiektu obserwowanego podczas wielu przyrostowych zmian mogą powodować pewne wady tego rozwiązania, jeśli koszt pojedynczej aktualizacji jest wysoki. W takim wypadku, konieczne jest zarządzanie powiadomieniami, aby obserwator nie był powiadamiany zbyt często lub zbyt szybko (Cooper, 2001).

Przykładem implementacji wzorca jest interfejs `Observer` i klasa `Observable` w pakiecie standardowym *Java.util* języka Java.

Pełnomocnik (ang. *Proxy*)

Wzorzec *Pełnomocnik* wykorzystywany jest do reprezentowania skomplikowanego obiektu lub obiektu, którego utworzenie wymaga dużego nakładu czasu lub dużego nakładu zasobów, za pomocą obiektu prostego, po to aby odłożyć czas utworzenia obiektu do momentu kiedy rzeczywiście obiekt ten będzie potrzebny. Rozwiązanie to wpływa nie tylko na poprawienie wydajności systemu, ale również na jego bezpieczeństwo, gdyż pozwala na zweryfikowanie uprawnień do obiektu przed jego użyciem.

W skład wzorca wchodzi interfejs `Subject` posiadający dwie implementacje. Pierwszą, w postaci obiektu głównego (`RealSubject`) posiadającego funkcjonalność wymaganą przez klienta. Oraz drugą, obiektu-pełnomocnika `Proxy` działającego w imieniu obiektu głównego i przechowującego część jego atrybutów. Dzięki czemu możliwe jest zastąpienie obiektu głównego obiektem `Proxy`, który realizuje niektóre żądania klienta w sposób kompletny, a inne żądania deleguje do obiektu głównego (`RealSubject`). Obiekt-pełnomocnik odpowiedzialny jest za utworzenie lub załadowanie do pamięci obiektu głównego. W szczególności obiekt `Proxy` może utworzyć obiekt `RealSubject` znacznie później niż żąda tego klient, a tym samym pozwala na oszczędność czasu i innych zasobów.



Rys. 4.15. Diagram klas wzorca Pełnomocnik

Źródło: Opracowanie własne na podstawie (Freeman, 2004)

Wyróżniane są cztery rodzaje wzorca *Proxy* definiujące jednocześnie sytuacje, w których mogą zostać wykorzystane (Source Making, 2011):

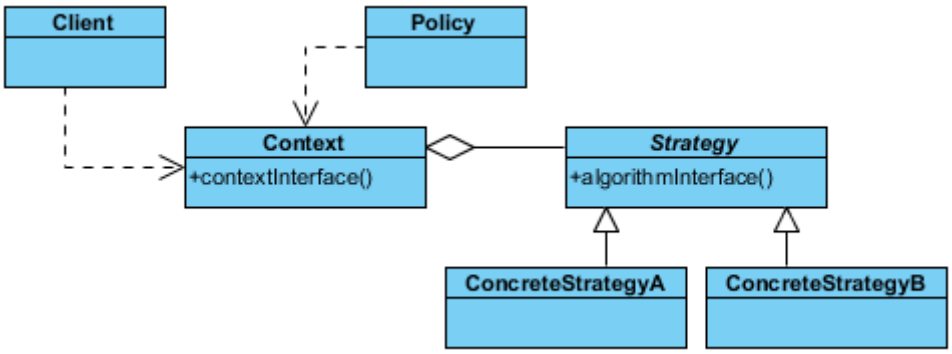
- **Zdalny obiekt Proxy** (ang. *Remote Proxy*). Służy do reprezentacji obiektu znajdującego się w innej przestrzeni adresowej, np. na innym komputerze. Dzięki jego wykorzystaniu dla lokalnych klientów wszystkie odwołania są pozornie lokalne. `Proxy` odpowiedzialny jest wtedy za zdalne wywołania metod poprzez sieć, serializację parametrów i odebranie wyników (np. CORBA lub RPC).
- **Wirtualny obiekt Proxy** (ang. *Virtual Proxy*). Zastępuje obiekt `RealSubject` o dużych wymaganiach zasobowych, np. alokujący duży obszar pamięci. Aby opóźnić (lub zastąpić) proces tworzenia takiego obiektu, `Proxy` obsługuje wszystkie zadania obiektu `RealSubject`, które nie wymagają odwołań do tego obszaru pamięci.
- **Ochronny obiekt Proxy** (ang. *Protective Proxy*). Kontroluje nieautoryzowany dostępu do obiektu `RealSubject`. Obiekt `RealSubject` nigdy nie jest bezpośrednio dostępny dla klientów; w ich imieniu `Proxy` określa, którym z nich możliwe jest udostępnienie usług oferowanych przez `RealSubject`.

- **Sprytne odwołanie** (ang. *Smart Proxy*). Pozwala na wykonanie dodatkowych akcji podczas dostępu do obiektu `RealSubject`, takich jak: zliczanie referencji do obiektu, sprawdzanie stanu obiektu, czy ładowanie obiektu do pamięci.

Przykładem wirtualnego obiektu `Proxy` może być obiekt `ProxyImage`, który działa w imieniu obiektu `Image` przechowywanego na dysku. `ProxyImage` przechowuje te same informacje co `Image` z wyjątkiem samego obrazu. Obiekt `ProxyImage` jest więc w stanie realizować wszystkie żądania z wyjątkiem operacji związanych z przetwarzaniem zawartości obrazu. W przypadku kiedy następuję żądanie wykonania operacji na samym obrazie, obiekt `ProxyImage` tworzy obiekt `RealImage` i ładuje rzeczywisty obraz z dysku.

Strategia (ang. *Strategy*)

Wzorzec *Strategia* rozdziela klasy wyznaczające reguły działania od algorytmów je realizujących, dzięki czemu algorytmy te mogą być wymieniane lub zmieniane w sposób niewidoczny dla klienta.



Rys. 4.16. Diagram klas wzorca Strategia
Źródło: Opracowanie własne na podstawie (Freeman, 2004)

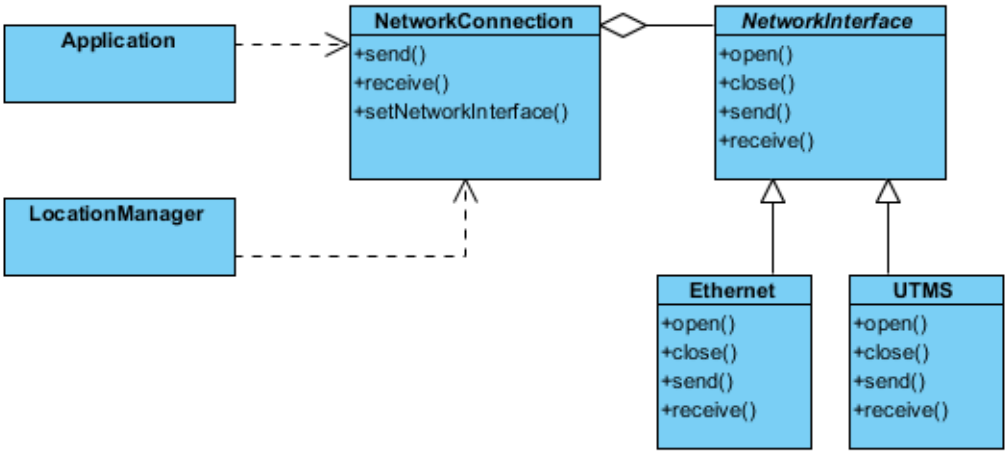
W koncepcji wzorca przedstawionego na rys. 4.16, klient korzysta z usług udostępnianych w danych kontekście (`Context`). Każda z tych usług realizowana jest za pomocą jednego z dostępnych algorytmów (`ConcreteStrategy`). Wspólny interfejs dla wszystkich algorytmów kontekstu opisuje abstrakcyjna klasa `Strategy`.

Natomiast klasa `Context` przełącza pomiędzy strategiami na podstawie decyzji obiektu `Policy`.

W przedstawionym wzorcu klasy algorytmów mogą być powiązane ze sobą poprzez dziedziczenie, bądź w przypadku implementacji wspólnego interfejsu mogą pozostać niezwiązane.

Wzorzec *Strategia* pozwala na dynamiczną zmianę algorytmu oraz dodawanie nowych bez modyfikacji klienta oraz klasy `Context`, przez co zwiększa elastyczność programu.

Wzorzec *Strategia* znalazł swoje zastosowanie m.in. w przełączaniu połączeń sieciowych w aplikacjach mobilnych (rys. 4.17). Aplikacja mobilna posiada zdolność wykorzystywania różnych połączeń sieciowych dostępnych dla urządzenia (sieć LAN, sieć bezprzewodowa, UMTS, itd.). Wybór konkretnego protokołu dokonywany jest automatycznie na podstawie aktualnego kontekstu użytkownika urządzania (lokalizacja, dostępność poszczególnych protokołów, koszt, moc sygnału oraz wiele innych). Wykorzystanie wzorca *Strategia* umożliwia w tym przypadku odseparowanie protokołów od wspólnego interfejsu sieciowego.



Rys.4.17. Zastosowanie wzorca Strategia do przełączania połączeń sieciowych w aplikacjach
Źródło: Opracowanie własne na podstawie (Bruegge, 2011)

4.5. WZORZEC MVC I MVP

Wzorzec **Model-Widok-Kontroler** (ang. *Model-View-Controller, MVC*) jest jednym z najczęściej opisywanych wzorców projektowych. Jest on wzorcem architektonicznym i, w przeciwieństwie do wcześniej zaprezentowanych wzorców, traktowany jest jako wzorzec złożony wykorzystujący ideę wzorców podstawowych.

Głównym założeniem tego wzorca jest podział aplikacji na trzy niezależne warstwy (rys. 4.18):

- Model – warstwę reprezentującą logikę biznesową.
- Widok – warstwę odpowiedzialną za prezentowanie informacji użytkownikowi.
- Kontroler – warstwę zarządzającą sekwencją interakcji z użytkownikiem.

Rys. 4.18. Wzorzec Model-Widok-Kontroler
Źródło: Opracowanie własne

Model reprezentuje wiedzę z dziedziny aplikacji, tzn. zarządza zachowaniem oraz stanem danych aplikacji. Odpowiada na żądania widoku dotyczących jego stanu i reaguje na polecenia zmiany stanu generowane przez kontroler. W danej aplikacji może istnieć kilka modeli tego samego elementu.

Widok jest odpowiedzialny za prezentację danych w obrębie graficznego interfejsu użytkownika. Posiada bezpośrednie referencje do modelu, z którego pobiera dane, gdy otrzymuje od kontrolera żądanie odświeżenia.

Z kolei kontroler odpowiedzialny jest za odbiór danych wejściowych od użytkownika, ich przetworzenie oraz analizę. Po przetworzeniu odebranych danych kontroler może wykonać następujące czynności:

- zmienić stan modelu,
- odświeżyć widok,
- przełączyć sterowanie na inny kontroler.

Każdy kontroler posiada bezpośrednie wskazania na określone modele i widoki, z którymi współpracuje. W aplikacji może istnieć jednocześnie wiele kontrolerów, ale tylko jeden z nich steruje aplikacją w danym momencie.

Wzorzec MCV jest wzorcem złożonym. Podstawowym sposobem rozbicia MVC na prostsze wzorce jest podział prezentowany przez (Gamma, 1994):

- Kompozyt w widoku — umożliwia tworzenie i pracę z zagnieżdżonymi widokami.
- Obserwator w modelu i widoku — umożliwia powiadamianie widoku przez model o zmianie jego stanu.
- Strategia w widoku i kontrolerze — widok jest obiektem skonfigurowanym jako kontekst strategii, którą jest kontroler.

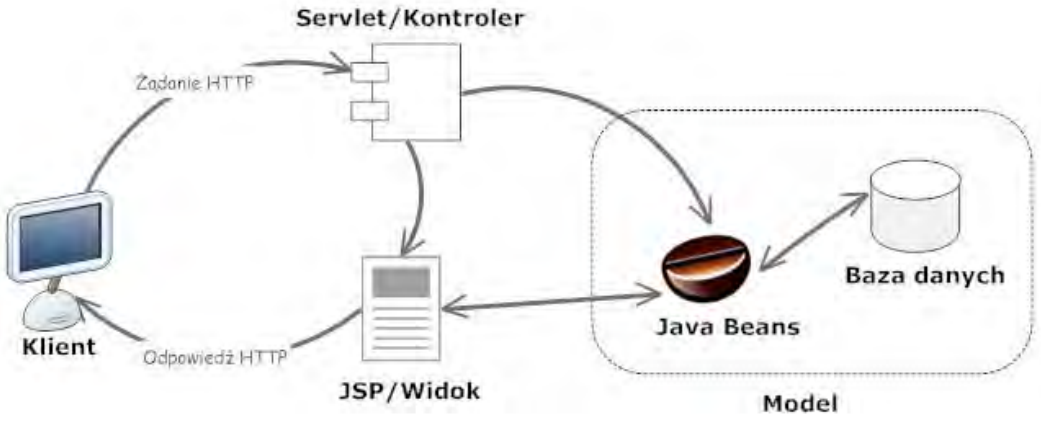
Przykładem wykorzystania wzorca architektonicznego MVC może być aplikacja internetowa zrealizowana w technologiach *Java Servlet* oraz *Java Server Pages* (JSP), której koncepcję zaprezentowano na rys. 4.19. Przedstawione technologie pozwalają na separację poszczególnych warstw aplikacji, w szczególności widok reprezentowany jest przez strony JSP, model stanowi baza danych wraz z komponentami *Java Beans*, w których postaci dane są przetwarzane, a rolę kontrolera pełnią serwlety przetwarzające żądania HTTP. Implementacja wzorca MVC w środowisku WWW jest znana również jako **Model-2**.

Zaletami wykorzystania wzorca projektowego MVC jest brak zależności modelu od widoków, dzięki czemu w aplikacji może współistnieć wiele widoków prezentujących te same dane. Biorąc pod uwagę także fakt, że warstwa prezentacji zmienia się o wiele częściej niż model, MVC umożliwia prostsze dodawanie oraz modyfikowanie istniejących widoków w aplikacji bez wpływu na logikę biznesową.

Z drugiej strony, wzorzec wprowadza dodatkową warstwę abstrakcji, co powoduje, że aplikacja jest trudniejsza do debugowania dla programistów. A ponieważ widoki są zależne od modeli, może to prowadzić do konieczności wprowadzenia kosztownych

zmian w widokach w przypadku przebudowy projektu modelu.

Wzorzec MVC stał się podstawą dla wielu innych wzorców. M.in. wzorcem pochodnym jest **Model-Widok-Prezenter** (ang. *Model-View-Presenter*, MVP).



Rys. 4.19. Wykorzystanie wzorca MVC w aplikacji opartej na Java Servlet i JSP
Źródło: Opracowanie własne na podstawie (Freeman, 2004)

4.6. WYBRANE WZORCE PROJEKTOWE W ZARZĄDZANIU ZASOBAMI

Spełnienie wymagań niefunkcjonalnych, a więc zapewnienie odpowiedniego poziomu wydajności, skalowalności, elastyczności, stabilności, czy też bezpieczeństwa systemu, zależy w dużej mierze od efektywnego zarządzania zasobami. Wzorce projektowe w zarządzaniu zasobami są pomocne w realizacji wymagań tego rodzaju. Z zestawu dostępnych wzorców, wybrano i omówiono trzy wzorce dla pozyskiwania zasobów (*Wzorzec wyszukiwania zasobów, Wzorzec leniwego pozyskiwania zasobów, Wzorzec chciwego pozyskiwania zasobów*) oraz jeden wzorzec dla zarządzania cyklem życia zasobów (*Wzorzec ponownego wykorzystania zasobów*).

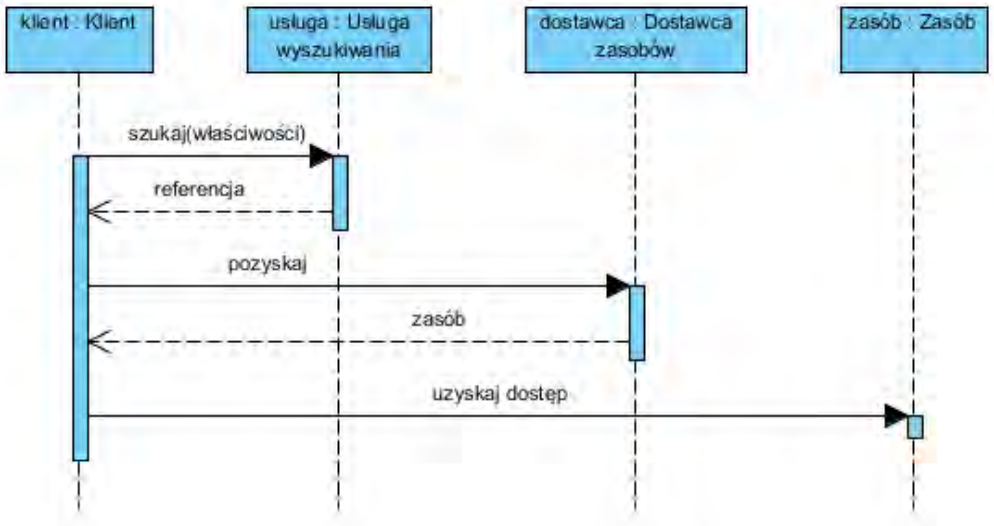
Wzorzec wyszukiwania zasobów (ang. *Lookup*)

Wzorzec wyszukiwania opisuje sposób odnajdywania i uzyskiwania dostępu do zasobów (lokalnych, jak i rozproszonych) poprzez zastosowania dodatkowej usługi

wyszukującej (ang. *Lookup Service*). Usługa ta pełni rolę pośrednika.

Dostawcy zasobów oferują jeden lub wiele zasobów różnych typów, a także mogą dodawać oraz usuwać oferowane zasoby, dlatego też klient (użytkownik zasobów) zmuszony jest do pozyskiwania bieżących informacji o oferowanych zasobach. Aby nie przeciążać systemu przesyłaniem komunikatów przez dostawców z informacją o aktualnie udostępnianych zasobach lub przez nowych klientów z pytaniem o oferowane w danym momencie zasoby, stosowane jest rozwiązanie polegające na udostępnieniu usługi wyszukiwania. Usługa wyszukiwania umożliwia dostawcom rejestrowanie zasobów poprzez reklamę (tj. dołączenie opisu właściwości zasobu), a klientom odnajdywanie interesujących ich zasobów za pomocą wyspecjalizowanego wyszukiwania na podstawie właściwości zasobu.

Implementację w*zorca wyszukiwania* przestawiono na diagramie sekwencji (rys. 4.20). Klient kieruje do usługi zapytanie dotyczące niezbędnego zasobu. W odpowiedzi usługa wyszukiwania odsyła klientowi referencję do zbioru dostawców oferujących poszukiwany zasób. Następnie klient zwraca się bezpośrednio do dostawców w celu pozyskania interesującego go zasobu.



Rys.4.20. Diagram sekwencji wzorca wyszukiwania zasobów

Źródło: Opracowanie własne na podstawie (Kircher, 2006)

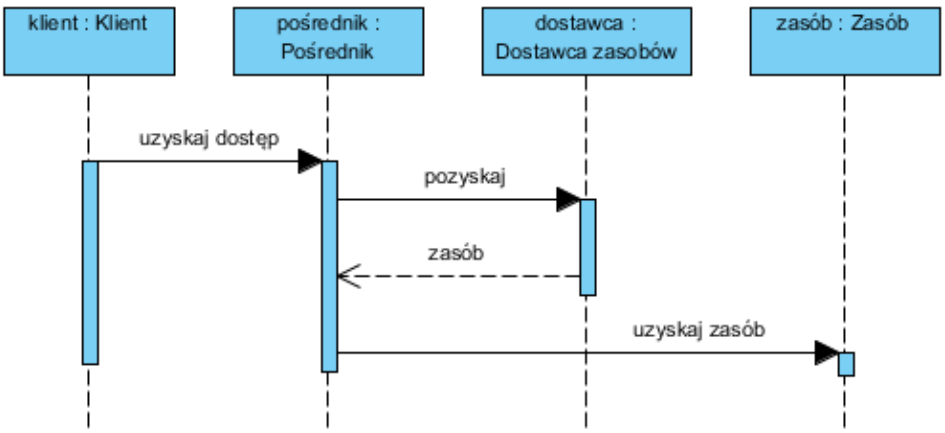
Przykładem wykorzystania *wzorca wyszukiwania* jest system złożony z wielu obiektów zdalnych zaprojektowanych za pomocą technologii *CORBA*, która publikuje informacje o punktach dostępu za pośrednictwem usługi nazewnictwa i odpowiednich właściwości lub plików konfiguracyjnych. Inny przykładem jest mechanizm działania usługi DNS w sieciach TCP/IP.

Zastosowanie *wzorca wyszukiwania* umożliwia określenie dostępności zasobów w danym momencie oraz gwarantuje ich właściwy dobór (według właściwości zdefiniowanych przez dostawców). Uniezależnia system od lokalizacji zasobów, a także ukrywa położenia zasobów przez klientami (użytkownikami zasobów). Dodatkowo, systemy zbudowane z wykorzystaniem usługi wyszukiwania wymagają niewielkich lub nie wymagają żadnych nakładów pracy w obszarze ich konfiguracji. Z drugiej strony, rozwiązanie to jest podatne na uszkodzenia, tzn. w przypadku awarii egzemplarza usługi wyszukiwania, system traci nie tylko zarejestrowane referencje, ale także powiązane z nimi właściwości.

Wzorzec leniwego pozyskiwania zasobów (ang. *Lazy Acquisition*)

Wzorzec leniwego pozyskiwania zasobów odkłada operację pozyskania zasobów na ostatni możliwy moment w czasie wykonywania aplikacji, tym samym optymalizując proces wykorzystywania zasobów. Wzorzec wykorzystywany jest w systemach z ograniczonymi zasobami, które muszą spełniać wysokie wymagania w zakresie przepustowości oraz dostępności. Pozyskiwanie zasobów już w czasie uruchamiania aplikacji mogłoby prowadzić do znaczących opóźnień w początkowych etapach pracy systemu, a także do marnotrawstwa zasobów.

Kluczowym elementem wzorca jest operacja pozyskania zasobu przez pośrednika, w momencie gdy klient (użytkownik zasobu) po raz pierwszy żąda dostępu do danego zasobu. Pośrednik zasobu na początkowo nie dysponuje własnym zasobem. Właściwe pozyskanie zasobu następuje tylko raz w odpowiedzi na pierwsze żądanie klienta (rys. 4.21). Natomiast kolejne żądania są już przekazywane przez pośrednika do faktycznego zasobu. Klient nie dysponuje wiedzą na temat dodatkowego poziomu interakcji – nie wie o istnieniu pośrednika w dostępie do zasobu.



Rys. 4.21. Diagram sekwencji wzorca leniwego pozyskiwania zasobów
Źródło: Opracowanie własne na podstawie (Kircher, 2006)

Z *wzorca leniwego pozyskiwania zasobów* wywodzi się wiele wyspecjalizowanych wzorców (Kircher, 2006). Przykładowo są to wzorce takie jak:

- *Leniwe tworzenie egzemplarzy* (ang. *Lazy Instantation*). Odkłada proces tworzenia egzemplarzy obiektów do momentu, w którym klient zażąda dostępu do tych egzemplarzy.
- *Leniwe wczytywanie* (ang. *Lazy Load*). Odkłada wczytywanie współdzielonych bibliotek do czasu, gdy zawarte w nich elementy programu będą rzeczywiście potrzebne.
- *Leniwy stan* (ang. *Lazy State*). Odkłada proces inicjalizacji stanu obiektu do czasu otrzymania odpowiedniego żądania dostępu.
- *Leniwe wyznaczania wartości* (ang. *Lazy Evaluation*). Wyrażenie nie jest obliczane do momentu, w którym jego wynik rzeczywiście jest potrzebny.

Przykładem wykorzystania odmiany *wzorca leniwego pozyskiwania zasobów* może być język Haskell umożliwiający, podobnie jak inne funkcyjne języki programowania, leniwe wyznaczanie wartości wyrażeń.

Stosowanie *wzorca leniwego pozyskiwania zasobów* minimalizuje ryzyko szybkiego wyczerpania zasobów, których może zabraknąć w momencie, gdy będą najbardziej potrzebne. Dodatkowo optymalizuje proces uruchamiania systemu, gdyż

jego mechanizm daje pewność, że zasoby nie potrzebne od razu będą pozyskane w późniejszym terminie. Niemniej stosowanie wzorca wiąże się z pewnymi utrudnieniami. Wprowadza on dodatkowe wymagania w zakresie wykorzystania pamięci (składowanie pośredników w dostępie do zasobów), a także może powodować znaczne opóźnienia w trakcie wykonywania programu. Dlatego też wzorzec leniwego pozyskiwania zasobów nie może być stosowany w systemach czasu rzeczywistego.

Wzorzec chciwego pozyskiwania zasobów (ang. *Eager Acquisition*)

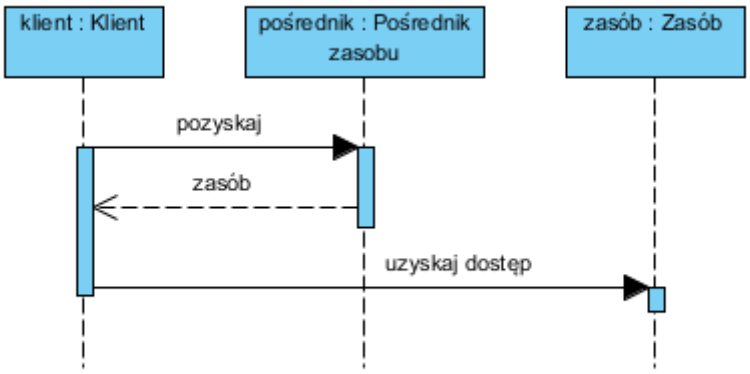
Przeciwieństwem *wzorca leniwego pozyskiwania zasobów* jest *wzorzec chciwego pozyskiwania zasobów.* Wzorzec ten podobnie do *wzorca leniwego pozyskiwania zasobów* definiuje sposób realizacji operacji pozyskiwania zasobów w czasie wykonywania aplikacji, ale działania pozyskiwania zasobów są realizowane w sposób przewidywalny i „chciwy" - zasoby są inicjalizowane jeszcze przed ich właściwym użyciem (Kircher, 2006).

Wzorzec chciwego pozyskiwania zasobów jest stosowany w systemach czasu rzeczywistego oraz innych systemach restrykcyjnie podchodzących do problemu wyboru sposobu i momentu pozyskania zasobów. Do takich systemów należą krytyczne systemy przemysłowe, wysoce skalowane aplikacje internetowe, aplikacje z graficznym interfejsem użytkownika.

Przed właściwym użyciem zasobów, a więc najczęściej w czasie uruchamiania aplikacji, zasoby są pozyskiwane od dostawcy poprzez specjalnego pośrednika zasobów (rys. 4.22). Pozyskane zasoby są składowane w efektywnym kontenerze, a wszystkie późniejsze żądania zasobów są przechwytywane przez pośrednika, który uzyskuje dostęp do kontenera i zwraca żądane zasoby. Sam czas pozyskiwania zasobów przez pośrednika można zrealizować stosując rozmaite strategie (np. w momencie uruchamiania systemu lub w czasie wykonywania programu) uwzględniające czynniki takie jak: moment faktycznego użycia zasobów, ilość zasobów, zależności pomiędzy zasobami oraz czas ich pozyskiwania.

Korzyściami, jakie niesie zastosowanie *wzorca chciwego pozyskiwania zasobów*, są: przewidywalna dostępność zasobów, brak zmiennych opóźnień w procesie pozyskiwania zasobów, elastyczność w zakresie zmian strategii pozyskiwania

zasobów, przeźroczystość procesu pozyskiwania zasobów dla klientów. Niemniej rozwiązanie to posiada również wady – istnieje możliwość nieuzasadnionego wyczerpania zasobów poprzez pozyskiwanie nadmiernej ilości zasobów. Utrudnione jest także efektywne zarządzanie pozyskanymi zasobami.



Rys. 4.22. Diagram sekwencji wzorca chciwego pozyskiwania zasobów
Źródło: Opracowanie własne na podstawie (Kircher, 2006)

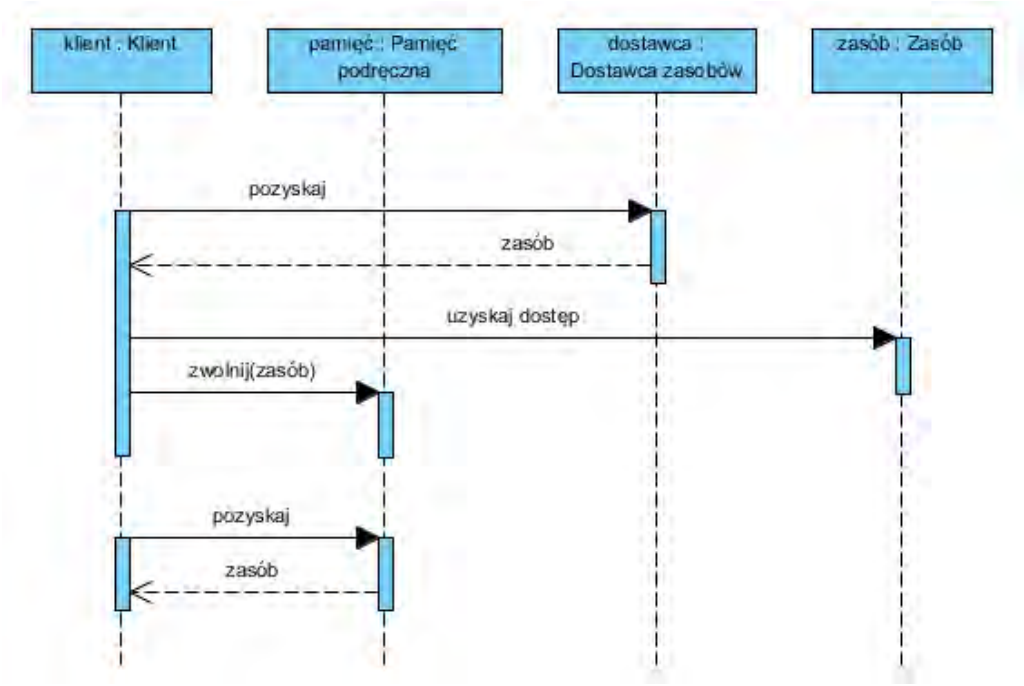
Wzorzec ponownego wykorzystania zasobów (ang. *Caching*)

Wzorzec ponownego wykorzystania zasobów określa sposób unikania kosztownych operacji ponownego pozyskiwania zasobów, ponieważ nie zwalnia zasobów zaraz po ich wykorzystaniu. Pozyskane zasoby, zachowując swój identyfikator, są składowane w buforze (nazywanym pamięcią podręczną) zapewniającym możliwie szybki dostęp w przypadku wystąpienia potrzeby ich ponownego wykorzystania (Grand, 1998).

We wzorcu klient uzyskuje dostęp do określonego zasobu u dostawcy. Następnie po jego wykorzystaniu, zasób zwracany jest do pamięci podręcznej zamiast do swojego dostawcy. W sytuacji, gdy klient ponownie musi uzyskać dostęp do tego samego zasobu, nie korzysta już z dostawcy, tylko pobiera ten zasób z pamięci podręcznej (rys. 4.23).

Przy implementacji wzorca *Caching* należy pamiętać o określeniu sposobu

integracji z pamięcią podręczną, wyborze strategii usuwania zasobów z pamięci podręcznej oraz zapewnieniu spójności pomiędzy oryginalnym zasobem zarządzanym przez dostawcę a kopią zasobu przechowywaną w pamięci podręcznej.



Rys. 4.23.Diagram sekwencji wzorca Caching
Źródło: Opracowanie własne na podstawie (Kircher, 2006)

Wzorzec *Caching* niweluje niepotrzebne opóźnienia związane z wielokrotnym pozyskiwaniem, inicjalizacją oraz zwalnianiem tego samego zasobu, ogranicza liczbę operacji, dzięki czemu system działa wydajnie, jest stabilny i skalowalny. Dodatkowo, wykorzystanie pamięci podręcznej zwiększa dostępność zasobów w przypadku, gdy oryginalni dostawcy są czasowo niedostępni. Natomiast do ut rudnieni stosowania wzorca można zaliczyć złożoność implementacji procesu synchronizacji stanu zasobów przechowywanych w pamięci i oryginalnych danych, które ten zasób reprezentuje.

Wzorzec *Caching* jest powszechnie stosowanym wzorcem przez systemy operacyjne np. do stronicowania - utrzymywane w pamięci podręcznej strony

pozwalają uniknąć kosztownych operacji odczytu dyskowego pliku wymiany. Innym przykładem wykorzystania wzorca są przeglądarki internetowe, które przechowują w pamięci podręcznej najczęściej otwierane strony www.

Podsumowując, wykorzystywanie wzorców projektowych w projektach realizowanych metodykami zwinnymi niesie ze sobą wiele korzyści zarówno w aspekcie technicznym, jak i organizacji pracy. Przede wszystkim, używanie gotowych rozwiązań pozwala programistom szybciej uporać się ze standardowymi problemami i gwarantuje, że system będzie bardziej elastyczny na w prowadzanie zmian i łatwiejszy w rozbudowie.

Dodatkowo korzystanie z wzorców powoduje upowszechnienie się standardowej terminologii z nimi związanej. Dzięki czemu współpraca między członkami zespołu może stać się bardziej efektywna.

Niemniej wybór wzorca projektowego, oraz jego przystosowanie do specyfiki konkretnego problemu, nie jest zadaniem łatwym i wymaga sporego doświadczenia od programisty.

4.7. PYTANIA KONTROLNE

1. Wyjaśnij czym są wzorce projektowe.
2. Podaj podstawową klasyfikację wzorców projektowych przedstawioną przez „bandę czworga”.
3. Jaki wzorzec należy zastosować w przypadku niekompatybilnych interfejsów klas?
4. Jaki wzorzec umożliwia reprezentowanie żądań wykonania operacji w formie obiektu?
5. Dlaczego wzorzec *Model-Widok-Kontroler* jest nazywany wzorcem złożonym?
6. Wymień i omów dwa przykładowe wzorce projektowe dla pozyskiwania zasobów.
7. Omów wzorzec *Caching* dla zarządzania cyklem życia zasobów.
8. Jakie korzyści daje wykorzystanie wzorców?

5

Refaktoryzacja kodu i inne techniki polepszania jego jakości

Cel

Rozdział ten poświęcony jest mechanizmom i technikom wspomagającym uzyskanie i utrzymanie kodu o wysokiej jakości. Zawiera on opis idei refaktoryzacji oraz wskazuje na podstawowe zasady jej przeprowadzania. W rozdziale znaleźć można też charakterystykę podstawowych błędów w kodzie oraz sposoby ich poprawy. Na końcu rozdziału opisane zostały narzędzia wspomagające ten proces.

Plan

1. Jakość kodu a zasady refaktoryzacji.
2. Brzydkie zapachy w kodzie, czyli typowe błędy.
3. Metody refaktoryzacji.
4. Narzędzia refaktoryzacji.

5.1. JAKOŚĆ KODU A ZASADY REFAKTORYZACJI

Kod źródłowy bardzo często zawiera błędy. Brak dokumentacji, brak spójności kodu, niezgodność ze strukturą aplikacji, niewystarczający zapis zdarzeń to tylko niektóre z długiej listy grzechów popełnianych przez programistów. Rozbudowa aplikacji, dodawanie kolejnych funkcjonalności oraz eliminacja błędów powoduje zwykle zmiany w strukturze kodu. Jeśli brakuje nadzoru nad spójnością powstającego kodu, jego jakość ulec może znacznemu pogorszeniu.

Słaba jakość kodu jest trudna do kontrolowania (Bijay, Patton, 2007). Spadek jakości kodu znacznie utrudnia pracę i możliwości jego rozbudowy, przekształceń oraz konserwacji. Słaba jakość kodu utrudnia współpracę programistów, powoduje wzrost czasu pracy i zwiększa podatność na błędy. Praca z niechlujnie napisanym kodem po jakimś czasie stanowi problem nawet dla jego autora, nie wspominając o programistach, którzy mają w kodzie dokonać zmian czy poprawek.

Na wydajną pracę z kodem mają wpływ wszystkie wymiary jego jakości wymienione w rozdziale 3 (Wydajność, Niezawodność, Wytrzymałość, Łatwość naprawy, Estetyka, Cechy funkcjonalne, Reputacja, Zgodność ze standardami). I choć praca nad dobrej jakości kodem wymaga nieco czasu i wytrwałości, to jej efekty z nawiązką zwrócą się w przyszłości.

Każdy zespół, a w szczególności zespół pracujący zgodnie z metodyką programowania zwinnego, powinien cyklicznie poprawiać jakość kodu. Proces przeglądu i poprawy kodu w taki sposób, aby uczynić go bardziej czytelnym, intuicyjnym i o wyższej jakości jest określany mianem refaktoryzacji. Refaktoryzacja w praktyce polega zwykle na wprowadzaniu serii drobnych poprawek, które poprawiają strukturę systemu jednak pozostają bez wpływu na logikę i przeznaczenie biznesowe (Martin, Martin, 2007). Zmiana struktury i zależności w kodzie ma na celu jego uproszczenie, ujednolicenie tak, aby był łatwiejszy do czytania, analizy i ewentualnej przebudowy w przyszłości.

Każda taka zmiana powinna mieć swoje uzasadnienie, ponieważ szereg przekształceń prowadzić może do poważnych zmian w projekcie. Zmiany takie mogą

łatwo wymknąć się spod kontroli. Drobna zmiana w jednej części kodu spowodować może poważne problemy w projekcie. Uwaga ta dotyczy szczególnie takich projektów, w których brakuje odpowiednich mechanizmów zabezpieczających przed niekontrolowanymi zmianami, jak choćby testy jednostkowe. Testy jednostkowe były omówione szerzej w rozdziale 3. W tym miejscu warto tylko przypomnieć, że testy powinny być wykonane nawet po drobnych transformacjach kodu aby wyeliminować niepożądane skutki uboczne. Po pomyślnym przejściu testów można rozpocząć wprowadzanie kolejnej poprawki, po której znów należy uruchomić zestaw testów (Skroban, 2010, s. 205). Ten sposób przeprowadzania procesu refaktoryzacji daje możliwość utrzymania działającego systemu mimo wprowadzanych zmian.

Kolejnym istotnym aspektem poprawiającym bezpieczeństwo procesu refaktoryzacji jest korzystanie z systemu kontroli wersji, który umożliwi wprowadzenie trwałych zmian do projektu dopiero po ich przetestowaniu i zatwierdzeniu.

Zatem jak często powinno się przeprowadzać refaktoryzację? Z pewnością refaktoryzacja na poziomie końca projektu czy końca iteracji nie ma sensu, gdyż byłaby zbyt czasochłonna a jej wyniki mogłyby być trudne do przewidzenia. Nawet jeden dzień roboczy bez refaktoryzacji może spowodować spore problemy. W praktyce refaktoryzacji dokonuje się cały czas, co godzinę, co pół godziny, po ukończeniu metody. Takie podejście „małych kroków” (Martin, Martin, 2007) zgodne jest z metodyką programowania zwinnego i tylko wtedy zespół może liczyć na to, że jakość tworzonego kodu jest zadowalająca i że zachowane są zasady dotyczące jego prostoty, czytelności i łatwości wprowadzania zmian.

Ewolucja oprogramowania

Refaktoryzacja nie jest powiązana bezpośrednio z językiem programowania czy aspektem programowania. Zarówno przy tworzeniu rdzenia i szkieletu aplikacji, jak i przy procedurach bazodanowych i skryptach systemowych obowiązują te same zasady przeprowadzania refaktoryzacji.

Wyrobienie sobie nawyku przeprowadzania refaktoryzacji kodu przyniesie ogromne korzyści w każdym projekcie i dla każdego zespołu, a czas na nią poświęcony nigdy nie będzie czasem straconym. Nawet, jeśli aplikacja nie będzie w przyszłości przebudowywana i rozwijana, to dobrze napisany kod chętniej będzie

wykorzystywany w kolejnych projektach i poszerzy zbiór bibliotek dobrego kodu. Jednak warto pamiętać, że moduł oprogramowania powinien spełniać trzy podstawowe aspekty:

- realizację funkcjonalności,
- gotowość na zmiany,
- komunikacja z „czytelnikami kodu".

Trudno się nie zgodzić, że drugi i trzeci aspekt dotyczy jakości kodu i silnie związanej z nim refaktoryzacji. I choć idea „Jeśli coś działa, nie naprawiaj tego" może na pozór wydawać się słuszna, to jeśli uwzględni się wszystkie trzy aspekty oprogramowania okaże się, że refaktoryzacja ma głęboki sens.

W rzeczywistości pierwsza wersja kodu aplikacji zawsze ulega zmianom. I choć znaczenie właściwego zarządzania i dobrego projektu trudno przecenić, to jednak zmiany zawsze się pojawiają niezależnie od specyfiki aplikacji i rodzaju klienta. Tworzenie kodu, jego sprawdzanie, debugowanie oraz pisanie testów jednostkowych zajmuje od 30 do 65 procent pracy przy projekcie (w zależności od rodzaju projektu) (McConnel, 2010). Im większy projekt, tym trudniej uzyskać zgodność z dokumentacją i uniknąć poważnych zmian w kodzie i strukturze. Nawet w dobrze zarządzanych projektach zmiany dotyczą od jednego do czterech procent wymagań miesięcznie (Jones, 2000) co oczywiście pociąga za sobą konsekwencje przebudowy kodu i zmiany dokumentacji. Ewolucja oprogramowania jest nieunikniona. Im wcześniej ten fakt zostanie przez zespół zrozumiany, tym szybciej zostanie wykorzystany w celu poprawy jakości kodu. Jednak nawet małe projekty pisane w pojedynkę warto refaktoryzować. Dbanie o jakość kodu to dobra praktyka.

Podczas przeprowadzania refaktoryzacji nie można też zapomnieć o samej technologii wytwarzania oprogramowania. Refaktoryzacja powinna objąć dostosowanie kodu do nowszych, bardziej wydajnych wersji oprogramowania. Warto wykorzystywać nowsze możliwości środowiska czy frameworka zamiast bezkrytycznie stosować stare procedury, które kiedyś uważane były za poprawne. Jednocześnie unikanie oddalania się od konwencji charakterystycznej dla danego języka jest jedną z podstawowych przesłanek tworzenia spójnego kodu.

Duże projekty mają często z góry narzucone standardy kodu, które są sprecyzowane jeszcze przed rozpoczęciem prac z kodem. Mimo to badania pokazują,

że w dużych projektach liczba błędów przypadających na tysiąc linii kodu jest większa niż w przypadku małych przedsięwzięć (przy założeniu, że pozostałe parametry wpływające na projekt pozostają stałe).

Ogólne zasady tworzenia i pielęgnacji kodu

Istnieje wiele metod programowania i wiele „dobrych praktyk" polecanych dla tworzenia dobrej jakości kodu. Nie ma potrzeby dążyć do spełnienia wszystkich możliwych zaleceń. Wystarczy dobrać takie, które będą pasować do wielkości i charakteru projektu. Wśród popularnych metod tworzenia i kontroli kodu dla metodyk lekkich można wyróżnić programowanie w parach, rozpoczynanie każdego nowego etapu pracy od tworzenia testów czy przeprowadzanie formalnych inspekcji kodu.

Istotne praktyki programistyczne można podzielić na trzy kategorie:

- zasady tworzenia dobrego kodu,
- zagadnienia kontroli jakości i testowania kodu,
- wybór narzędzi wspomagających pracę.

Zasady tworzenia dobrego kodu określają praktyki (standardy) dotyczące wszystkiego, co związane jest z pisaniem (lub uzyskiwaniem poprzez refaktoryzację) wydajnego kodu. Do najważniejszych należy zaliczyć następujące:

- Określenie zasad komentowania oraz używania samodokumentującego się kodu.
- Zasady dotyczące tworzenia klas i metod zorientowanych wyłącznie na zadanie, do którego zostały stworzone; komplikowanie ich, przesadne wydłużanie i „wzbogacanie" o dodatkowe zadania nie jest nigdy pożądane i prowadzi do komplikacji. O wiele lepiej jest stworzyć więcej mniejszych klas/metod (Martin, 2010).
- Warunki, jakie musi spełniać kod, aby mógł być ponownie wykorzystany. Do takich warunków może należeć enkapsulacja, czy narzucona wcześniej odpowiednia struktura kodu. Jeśli kod ma być wykorzystany w przyszłości przez innych programistów istotna także jest jego czytelność i przejrzystość, a także komentarze utrzymane w określonej wcześniej konwencji.
- Sposoby obsługi błędów oraz standardy zabezpieczeń przed niekontrolowanym wpływem zmian na projekt. Stworzenie scentralizowanego mechanizmu

komunikatów oraz narzucenie standardów pracy z wyjątkami w całym projekcie (w tym zdefiniowanie dopuszczalnego poziomu abstrakcji wyjątków) usystematyzuje poziom bezpieczeństwa i usprawni proces zarządzania zmianami.

- Ustalenie zasad współpracy z członkami zespołu w projekcie; programowanie w parach, określenie zasad współdzielenia kodu, kontrole jakości (czytanie) kodu, formalne i nieformalne inspekcje – każda z metod programowania zespołowego jest skuteczna, jeśli konsekwentnie się ją stosuje (Martin, Martin, 2007); ważne jest też dostosowanie metody do umiejętności, doświadczenie i cech charakteru członków grupy.
- Wprowadzenie nacisku na optymalizację oraz korzystanie z konstrukcji ułatwiających i przyspieszających działanie. Odkrywanie koła na nowo nigdy nie jest metodą przynoszącą korzyści; o wiele lepiej jest poświęcić chwilę na poszukiwanie i analizę rozwiązania problemu, którą ktoś już wcześniej się zajął. W ten sposób nie tylko można uniknąć problemów z brakiem spójności i optymalizacją, ale także nauczyć się nowych technik i innego spojrzenia na zagadnienie.
- Określenie procedur integracji wskazujących na warunki, które kod powinien spełnić aby mógł zostać dołączony do głównego repozytorium.

Zagadnienia kontroli jakości i testowania kodu które nie są związane z bezpośrednim pisaniem kodu, ale ze wszystkim tym, co go otacza:

- Określenie, czy przed rozpoczęciem kolejnego etapu z kodem będą tworzone testy. Testy jednostkowe ułatwiają zachowanie spójności i jakości kodu. W dużych projektach należy z góry ustalić kto będzie odpowiedzialny za tworzenie testów i których modułów będą one dotyczyć. Dobrej jakości kod powinien być przynajmniej w 80% pokryty testami.
- Określenie, czy przed dołączeniem kodu do repozytorium należy przeprowadzić testy integracyjne.
- Ustalenie zasad kontroli kodu i metod jego weryfikacji przez innych członków zespołu.

Wybór narzędzi wspomagających pracę jest również zagadnieniem wartym przemyślenia. Poza oczywistymi kwestiami takimi jak platforma programistyczna, język czy wersja kompilatora warto zastanowić się również nad:

- Narzędziem kontroli wersji adekwatnym do rozmiary projektu i wielkości zespołu.
- Narzędziami wspomagającymi zarządzanie projektem.
- Metodykami nadzoru nad jakością oprogramowania i dokumentacji.
- Dopuszczeniem lub nie niestandardowych właściwości języka oraz pisaniem kodu w wielu językach.
- Dostosowaniem edytora, debuggera, mechanizmu sprawdzającego składnię oraz narzędzia do analizy statycznej kodu.
- Zaopatrzeniem się w narzędzie wspomagające refaktoryzację kodu.
- Skonfigurowaniem systemu testowania.

Podsumowując, warto wyrobić sobie dobry nawyk dbania o jakość kodu. Warto korzystać z metod i narzędzi wspomagających refaktoryzację, czyli poprawę tej jakości, jednak należy wybrać je przed rozpoczęciem pracy nad oprogramowaniem. Tworzenie dobrego kodu zawsze się opłaci ponieważ raz napisany kod w zasadzie zawsze ulega większym lub mniejszym zmianom w trakcie trwania projektu. Elementy dobrej jakości kodu łatwiej zostaną zaadoptowane do kolejnych projektów. Bo co może stanowić lepszą wizytówkę programisty niż czytelny, łatwy do zmiany, poprawnie i optymalnie stworzony kod.

5.2. BRZYDKIE ZAPACHY W KODZIE, CZYLI TYPOWE BŁĘDY

Istnieje zdefiniowany zestaw typowych błędów, które spotkać można w kodzie. Ich zdefiniowanie i wyodrębnienie w praktyce i opisanie w literaturze skutkuje gotowym zestawem procedur i dobrych praktyk, których zastosowanie się zaleca. Oczywiście tradycyjnie ze zbioru dobrych praktyk warto wybrać te, które najlepiej pasują do wymagań projektu.

Brzydkie zapachy w kodzie są błędami, których poprawa ma istotne znaczenie dla jakości i uniwersalności kodu.

Klasy, w których jest więcej błędów niż w pozostałej części projektu lub takie,

których kod wydaje się zawiły to problemy, które będą skutkować koniecznością przebudowy poprzedzonej długim debugowaniem. Dlatego często przeprowadzana refaktoryzacja powinna obejmować kilkanaście poniżej opisanych aspektów. Należą do nich: zbytnia zależność pomiędzy modułami, brak spójności oraz niezgodność poziomu abstrakcji poszczególnych elementów programu, nadmiarowe wykorzystywanie relacji dziedziczenia, powtórzenia fragmentów kodu oraz pozostawianie niewykorzystywanego kodu. Bardziej złożone zagadnienia, takie jak błędy w budowie klas, metod, zmiennych czy pętli opisane zostały z podziałem na poszczególne aspekty. W podobny sposób przedstawione zostały kwestie dotyczące zachowania bezpieczeństwa, ujednolicania nazewnictwa, sposobu komentowania oraz strukturyzacji kodu programu.

Zależność (powiązanie)

Ścisła zależność pomiędzy elementami kodu (klasy, procedury) nie jest zjawiskiem pożądanym. „Luźne powiązania" pomiędzy modułami wpierają elastyczność i powodują, że użycie jednego modułu przez drugi jest łatwe. Im niższy jest wprowadzony poziom zależności, tym lepiej.

Procedura z nagłówkiem `GetInstance(var1)` ma niższy poziom zależności niż procedura `InitArgs(var1,var2,var3,…,var15)`, której długa lista parametrów wymusza na module, który ją uruchomi bardzo dobrej znajomości szczegółów implementacji procedury. Im mniejsza jest liczba parametrów procedury i metod klasy, tym mniejsza jest ich zależność od wywołujących je modułów (Martin, 2010).

Kolejnym problemem jest użycie tych samych zmiennych globalnych w kilku modułach. Zmienne globalne stanowią połączenia ukryte co może uzależnić od siebie moduły w sposób niekontrolowany. Łączenie modułów powinno się wykonywać jedynie poprzez parametry, przy czym im mniejsza jest ich liczba, tym lepiej. Spośród możliwych połączeń pomiędzy modułami najgorsze są te, które wymagają od jednego modułu znajomości działania drugiego i jego praca opiera się na założeniu, że zależny moduł dokonał zakładanych czynności. Oczywiście podejście takie łatwo powoduje powstanie trudnych do wykrycia błędów. Taki problem określany jest jako Zbytnia intymność. Prowadzi on do złamania idei hermetyzacji, która zakłada, że klasy nie

muszą znać szczegółów swoich implementacji, ponieważ wystarcza im ich interfejs.

Poprawne powiązania modułów (w tym wypadku klas) powinny być skonstruowane na zasadzie hierarchii, rozpoczynając od najbardziej abstrakcyjnej (ogólnej) warstwy na górze i schodząc do bardziej szczegółowych reprezentacji pojęć.

Niespójność

Spójność, swoistość w programowaniu są czasem określane jako kohezja, czyli stopień koncentracji na głównym celu. Im wyższy poziom kohezji ma moduł (klasa lub metoda), tym łatwiej jest opanować złożoność kodu oraz opanować szczegóły jej działania. Każda klasa powinna mieć też ściśle określone odpowiedzialności (Martin, 2010).

Zagadnienie braku spójności jest powiązane z powiązaniem (zależnością) oraz problemami z zachowaniem stałego poziomu abstrakcji w projekcie.

Przykładem może być zmiana metody w klasie opisującej czcionkę (Listing 5.1).

Listing 5.1. Przykład braku spójności

```
class currentFont {
        …
        SetFontParams(sizeInPixels,isBold,isItalic,fontType);
        …
  }
```

W przykładzie klasa currentFont posiada metodę ustawiającą jednocześnie wszystkie atrybuty czcionki. Aby ją wywołać, moduł wywołujący musi określić wszystkie parametry niezależnie od tego, ile parametrów ma zostać zmienionych. Lepszym rozwiązaniem byłoby wydzielenie metod odpowiedzialnych za poszczególne parametry (Listing 5.2).

Listing 5.2. Poprawiony przykład z Listingu 5.1

```
class currentFont {
        …
        SetSizeInPixels(sizeInPixels);
        SetItalicOn();
        SetItalicOff();
        SetBoldOn();
        SetBoldOff();
        SetFontType(fontType);
        …
    }
```

Problem spójności jest także powiązany z problemem „wędrujących danych", pojawiającym się wtedy, gdy dane przekazywane są do jednej metody po to, aby ta przekazała je dalej. W takim przypadku warto sprawdzić zgodność poziomów abstrakcji.

Nieodpowiedni poziom abstrakcji

Abstrakcja jest jedną z podstawowych cech obiektowości. Jej ideą jest upraszczanie rzeczywistości i umożliwianie obserwowania rzeczywistości w uproszczonej formie. Abstrakcją ukrywającą implementację klasy jest jej interfejs, którego zadaniem powinno być grupowanie metod powiązanych ze sobą.

Klasa, która reprezentuje źle dobrany poziom abstrakcji jest zwykle zbiorem metod, które są luźno ze sobą powiązane (McConnel, 2010). Klasy takie są zwykle bardziej rozbudowane a poruszanie się po nich jest skomplikowane.

Refaktoryzacja takich klas polega na określeniu odpowiedniego poziomu abstrakcji, zgrupowaniu uzupełniających się usług i przeniesieniu ewentualnych niepowiązanych informacji i metod do innych klas. Należy także uważać, aby nie stracić uzyskanej spójności podczas ewolucji kodu.

Przykład (Listing 5.3) przedstawia klasę, której poziom abstrakcji nie jest właściwy. W jednej klasie dodane są metody dotyczące pojedynczych książek, zarządzania ich zbiorem jak również zbiorem autorów i bibliotekarzy.

Listing 5.3. Przykład klasy z niewłaściwym poziomem abstrakcji

```
class Library {   //spis książek
    public:
        …
        void AddBook(Book book);
        void RemoveBook(Book book);
        Book NextBook();
        Book PreviousBook();
        void ClearBooksList();

        void AddLibrarian(Librarian librarian);
        void RemoveLibrarian(Librarian librarian);

        void AddAuthor(Author author);
        void RemoveAuthor(Author author);
        void AssignAuthorToBook(Book book, Author author);
    }
```

O wiele lepiej będzie rozbić tę klasę na kilka mniejszych klas, z których każda będzie miała swój własny poziom abstrakcji (Listing 5.4).

Listing 5.4. Poprawiony przykład z Listingu 5.3

```
class currentFont {
    public:
        …
        SetSizeInPixels(sizeInPixels);
        SetItalicOn();
        SetItalicOff();
        SetBoldOn();
        SetBoldOff();
        SetFontType(fontType);
        …
    }

class Library {   //spis książek
    public:
        …
        void AddBook(Book book);
        void RemoveBook(Book book);
        Book NextBook();
        Book PreviousBook();
        void ClearBooksList();
    }
```

```
class Librarian { // bibliotekarz
    public:
        …
        void AddLibrarian(Librarian librarian);
        void RemoveLibrarian(Librarian librarian);
    }

class Author {    //autor
    public:
        …
        void AddAuthor(Author author);
        void RemoveAuthor(Author author);
    }

class Book {    //książka
    public:
      …
      void AssignAuthorToBook(Book book, Author author);
    }
```

Wielodziedziczenie i inne problemy z dziedziczeniem

Dziedziczenie określić można jako budowanie klas, które są specjalizacją innych klas. Klasa bazowa określa wspólne elementy dla klas pochodnych, co umożliwia unikanie powtórzeń kodu i w rezultacie jego upraszanie. W praktyce dziedziczenie publiczne implementuje zwykle relację „jest" (nowa klasa czy interfejs jest bardziej specjalistyczną wersją wersji bazowej).

Dziedziczenie jest jednym z podstawowych narzędzi walki ze złożonością. Jednak niekontrolowane użycie tej techniki może dać skutek odwrotny. Używając dziedziczenia należy uważać na kilka aspektów (Freeman i in., 2005):

- Należy unikać dziedziczenia tam, gdzie nie jest ono wymagane. W szczególności nie należy doprowadzać do sytuacji, gdy dana klasa bazowa ma tylko jedną klasę pochodną.
- Ponowne użycie nazw nieprzesłanianych metod klasy bazowej w klasie pochodnej jest zbieżnością nazw.
- Używanie metod pustych dla przysłonięcia metod klasy bazowej jest złą praktyką prowadzącą do niejasności i logicznych komplikacji.

- Należy unikać sytuacji, w której klasy pochodne przeciążające metodę klasy bazowej w sposób, który narusza jej kontrakt (tzw. odrzucony spadek).
- Należy unikać głębokich drzew dziedziczenia (im więcej jest poziomów dziedziczenia, tym łatwiej o błędy).
- Wielodziedziczenie jest niebezpieczną praktyką, która powoduje złożoność i trudności z przewidzeniem działania stworzonego kodu .
- Należy unikać sytuacji, w których klasy bazowe zależą od swoich klas pochodnych. Przeczy to regułom polimorfizmu i enkapsulacji dla programowania obiektowego.
- Warto stosować wspomnianą już zasadę Liskov (ang. *Liskov Substitution Principle. LSP*) – „Musi być możliwe korzystanie z podklas poprzez interfejs klasy bazowej bez konieczności posiadania wiedzy o tym, że używana jest klasa pochodna" (Hunt, Thomas, 2000; Fowler i in, 2000; McConnel, 2010).

Podczas tworzenia i refaktoryzacji dziedziczenia warto uwzględnić następujące zasady przytoczone w książce „Kod doskonały" (McConnel, 2010):

- Jeśli klasy mają wspólne dane, a nie zachowania, to należy utworzyć obiekt zawierający te klasy.
- Jeśli klasy mają wspólne zachowania, ale nie dane, to należy utworzyć klasę bazową definiującą wspólne metody.
- Jeśli klasy mają wspólne i dane, i zachowania, to należy utworzyć klasę bazową definiującą wspólne metody i dane.

Martwy kod

Fragmenty kodu, które nie są wykonywane, nie tylko przeszkadzają i utrudniają czytelność. Często znajdują się one w obsłudze wyjątków, które nigdy nie są wywoływane, w instrukcjach warunkowych lub niewywoływanych metodach. Wpływają też negatywnie na wprowadzanie zmian i konserwacji kodu. Taki kod szybko się dezaktualizuje, po pewnym czasie może nie być zgodny z zasadami lub założeniami. Dlatego też warto zadbać o jego usunięcie (Fowler i in, 2000).

Powielany kod (powtórzenia)

Powtórzenia kodu są w większości wypadków skutkiem niedostatecznego podziału na moduły. Największym problemem z powtórzonym kodem jest to, że jego modyfikacja wymaga działań w wielu miejscach. Powtórzony kod warto wyodrębnić i umieścić uogólnioną jego wersję w metodzie klasy bazowej, a metody specjalizowane – w podklasach (Hunt, Thomas, 2000).

Listing 5.5. Przykład krótkiej konstrukcji, która mogłaby się powtarzać w kilku miejscach

```
avgCost=totalCost/ordersNumber;
```

Listing 5.6. Zastąpienie konstrukcji z Listingu 5.5 prostą metodą

```
class Library {   //spis książek
    public:
        …
        void AddBook(Book book);
        void RemoveBook(Book book);
        Book NextBook();
        Book PreviousBook();
        void ClearBooksList();

        void AddLibrarian(Librarian librarian);
        void RemoveLibrarian(Librarian librarian);

        void AddAuthor(Author author);
        void RemoveAuthor(Author author);
        void AssignAuthorToBook(Book book, Author author);
    }
double AvgOrderCost(double totalCost, int ordersNumber) {
        if (ordersNumber~=0)
          {
          avgCost=(double)totalCost/ordersNumber;
          }
        else
          {
          avgCost=0;
          }
```

Nawet krótkie, jedno- lub dwulinijkowe konstrukcje powtarzane w kilku miejscach warto zastąpić metodą. Obliczenia np. (Listing 5.5) można zastąpić prostą metodą, która dodatkowo zabezpiecza przed błędną operacją (Listing 5.6).

Za dużo informacji

Poprawnie zdefiniowane klasy (lub zbiory klas) mają małe interfejsy, które umożliwiają wykonanie potrzebnych operacji. Źle zdefiniowane interfejsy są zbyt głębokie i mają zbyt wiele funkcji, co wymaga tworzenia skomplikowanych konstrukcji nawet do prostych operacji. Im mniej metod i zmiennych ma klasa i im mniej zmiennych metoda, tym lepiej. Ułatwia to ukrywanie danych i funkcji oraz eliminację zbędnych sprzężeń.

Zazdrość o kod

Poprawnie napisany kod zakłada, że klasa lub metoda korzysta przede wszystkim ze zmiennych i metod należących do tej klasy (Fowler i in, 2000). Błędem jest, jeśli metoda nadmiernie wykorzystuje zmienne i metody innej klasy, w szczególności do manipulowania danymi obiektu tej klasy. Warto się w takim przypadku zastanowić się, czy nie powinna być ona przeniesiona do tej innej klasy. Zasady czystości kodu to zalecają.

Aspekty bezpieczeństwa

Bezpieczeństwo aplikacji odgrywa często kluczową rolę w projekcie. Wymagania dotyczące zabezpieczeń są też zwykle określane w specyfikacji. Jednak poza spełnianiem tych ograniczeń i wymagań konieczne jest zapewnienie odpowiedniej odporności aplikacji na popełnione przez użytkownika błędy, niepoprawne dane wejściowe czy nieobsłużone wyjątki. Problemy z bezpieczeństwem można podzielić na kilka grup.

Zdjęte zabezpieczenia

Zdarza się, że podczas tworzenia kodu zapomina się lub odkłada na później dodanie zabezpieczeń. Podczas refaktoryzacji nie można pominąć sprawdzania poziomu

zabezpieczeń kodu. Istnieje kilka metod obsługi błędów, które można stosownie do potrzeb łączyć (zwracanie wartości neutralnej lub najbliższej dopuszczalnej wartości, powtarzanie wcześniej otrzymanych wyników, rejestrowanie ostrzeżeń w pliku, zwracanie kodów błędów, lokalna obsługa błędów, wywoływanie obsługi błędu, wyświetlanie komunikatów błędów, przerwanie pracy programu). Wybór metod jest duży i łatwo jest stracić spójność reguł, co z kolei może przyczynić się do utraty odporności kodu. Warto zawsze sprawdzać, jaką wartość zwraca funkcja – nawet, jeśli błąd nie jest spodziewany.

Niepoprawna obsługa wyjątków

Warto również zwrócić uwagę na sprawdzenie poprawności obsługi wyjątków. Powinny być one wykorzystywane jedynie do obsługi sytuacji naprawdę wyjątkowych. Należy unikać zgłaszania wyjątków w konstruktorach i destruktorach a ich obsługę należy przeprowadzać na odpowiednim poziomie abstrakcji (podobnie jak ma to miejsce w przypadku reguł zachowania poziomu abstrakcji dla klas i metod).

Przykład z Listingu 5.7 przedstawia nieprawidłowo obsłużony wyjątek. Pusty blok `catch` powoduje, że tracona jest informacja o problemie. Informacja ta nie jest zapisywana do logów, wyjątek nie jest też odpowiednio obsłużony. Dodatkowo obsługiwany wyjątek nie jest wystarczająco szczegółowy – nie spełnia zasady zachowania poziomu abstrakcji.

Listing 5.7. Przykład nieprawidłowo obsłużonego wyjątku

```
public User GetUserFromSQLDB() {
        …
        try {
          …
          // duża ilość logiki
          …
        } catch (Exception) {
        }
        …
    }
```

Wersja poprawiona (Listing 5.8) obsługuje konkretny rodzaj błędu (tu: połączenie z bazą danych). Po za tym informacja o obsługiwanym wyjątku jest zapisywana.

Listing 5.8. Poprawiony przykład z Listingu 5.7

```
public User GetUserFromSQLDB() {
        …
        try {
          …
          // duża ilość logiki
          …
        } catch (SQLException ex) {
          LogError("Nieoczekiwany wyjątek:"+ex.Message);
        }
        …
    }
```

Planując zasady obsługi wyjątków warto zbudować globalny moduł obsługi wyjątków (McConnel, 2010), który będzie wychwytywał wyrzucone przez aplikację wyjątki i informacje o nich zapisywał w jednym miejscu.

Nieuporządkowane instrukcje warunkowe

Obsługa błędów realizowana jest często w instrukcjach warunkowych. Warto je porządkować tak, aby po `if` zawsze była wykonywana znacząca (typowa) instrukcja. Nie powinno się również mieszać instrukcji znaczących z obsługą błędów jak również używać pustych instrukcji (Listingu 5.9).

Listing 5.9. Przykład nieprawidłowo skonstruowanych instrukcji warunkowych

```
enum Status {
        Success,
        Error
    }
enum ErrorStatus {
        None,
        PrintDataError,
        ProcessDataError,
        ReadDataFromCatalogError,
        OpenCatalogError
    }

public Class() {
        OpenCatalog(catalog, status);
        if (status == Status.Error) //błąd otwarcia katalogu
        {
          error = ErrorStatus.OpenCatalogError;
        else  //udane otwarcie katalogu
        {
          ReadDataFromCatalog(catalog, data, status);
          if (status == Status.Succes) //udany odczyt
          {
            ProcessData(data, processData, status);
            if (status == Status.Error) ) // błąd przetwarzania
              error = ErrorStatus.ProcessDataError;
            else  //udane przetwarzanie
            {
              PrintData(processData);
              if (status == Status.Error) //błąd wyświetlania
              {
                error = ErrorStatus.PrintDataError;
              }
              else       //udane wyświetlenie
              {
                UpdateData(processData);
                error = ErrorStatus.None;
              }
            }
           }
           else          //błąd odczytu
           {
             error = ErrorStatus.ReadDataFromCatalogError;
           }
         }
       }
```

O wiele lepiej będzie oddzielić przypadki normalne i błędne, tak, aby główna ścieżka działania była widoczniejsza (Listing 5.10).

Listing 5.10. Poprawiony przykład z Listingu 5.9

```
enum Status {
        Success,
        Error
    }

enum ErrorStatus {
        None,
        PrintDataError,
        ProcessDataError,
        ReadDataFromCatalogError,
        OpenCatalogError
    }

public Class() {
        OpenCatalog(catalog, status);
          if (status == Status.Succes)
                          //udane otwarcie katalogu
          {
            ReadDataFromCatalog(catalog, data, status);
            if (status == Status.Succes) //udany odczyt
            {
              ProcessData(data, processData, status);
              if (status == Status.Succes) )
                           //udane przetwarzanie
              {
                PrintData(processData);
                if (status == Status.Succes)
                              //udane wyświetlenie
                {
                  UpdateData(processData);
                  error = ErrorStatus.None;
                }
                else          //błąd wyświetlenia
                  error = ErrorStatus.PrintDataError;

              }
              else      //błąd przetwarzania
              {
                error = ErrorStatus.ProcessDataError;
              }
            }
```

```
          else         //błąd odczytu
           {
             error = ErrorStatus.ReadDataFromCatalogError;
           }
         }
         else     //błąd otwarcia katalogu
         {
           error = ErrorStatus.OpenCatalogError;
         }
   }
```

Zwracanie kodów błędów

Zwracanie kodów błędów nie jest dobrą praktyką (Martin, 2010). Takie podejście powoduje głębokie zagnieżdżenia, zaciemnia kod i powoduje konieczność natychmiastowej obsługi. Przykład takiego kodu prezentuje Listing 5.11.

Listing 5.11. Przykład nieprawidłowo obsłużonego wyjątku

```
if(deleteFileReference(ref)==S_OK{
  if(deleteFile(file)==S_OK){
    if (deleteClient(client)==S_OK{
      logger.log("klient usunięty");
    } else {
      logger.log("usuwanie klienta nieudane");
    }
  } else {
    logger.log("usuwanie pliku nieudane");
  }
} else {
  logger.log("usuwanie nieudane");
  return S_ERROR;
}
```

Prościej jest użyć instrukcji `try-catch` (Listing 5.12).

Listing 5.12. Poprawiony przykład z Listingu 5.11

```
try{
  deleteFileReference(ref);
  deleteFile(file);
  deleteClient(client);
}
catch (Exception e) {
      logger.log(e.getMessage());
}
```

Używanie instrukcji goto

Instrukcja goto, która ma swoje odpowiedniki w postaci wielu wyjść z metody lub pętli, nazywanych wyjść z pętli, mechanizmów przetwarzania błędów i obsługi wyjątków.

Analizy pokazują (Dikstra, 1968), że liczba instrukcji goto jest odwrotnie proporcjonalna do jakości kodu. Jest to spowodowane szeregiem cech, które instrukcja ta posiada. Przede wszystkim jej wpływ na czytelność, spójność i logiczną strukturę oraz sterowanie jest destrukcyjny. Dodatkowo utrudnia lub nawet uniemożliwia ona także wykorzystywanie mechanizmów optymalizacji (ze względu na to, że instrukcje mogą być od siebie mocno oddalone). Użycie goto może także prowadzić do spowolnienia programu i jego skomplikowania.

Kod stworzony „na zapas”

Kod taki powstaje często jako pozostałość po refaktoryzacji lub po przebudowanych procedurach. Programiści zostawiają go, bo uważają, że być może przyda się w przyszłości. Taka praktyka jednak nie jest dobra, ponieważ prowadzi do powstania błędów, powtórzeń i straty czasu podczas wprowadzania zmian. Takie fragmenty kodu w przyszłości zostaną potraktowane jako kod poprawny, pełniący konkretną funkcjonalność. Poza tym, rzadko się zdarza, aby programista „trafił” w przyszłe wymagania projektu.

Klasy

Klasy stanowią podstawę programowania obiektowego. Dlatego ich pielęgnacja i refaktoryzacja jest konieczna. Szczególnie popularne problemy są przedstawione poniżej.

Duża liczba metod

Im mniejsza jest liczba metod klasy, tym mniejsze jest prawdopodobieństwo popełnienia błędu. Duża liczba metod w klasie komplikuje jej strukturę i utrudnia posługiwanie się obiektami tej klasy.

Zwielokrotnienie wyjściowe

Zbyt duża liczba wywoływanych procedur powoduje wzrost złożoności i przeczy regule minimalizowania zależności.

Duża liczba pośrednich odwołań do innych klas

Połączenia pośrednie (np. `auto.Producer().Contact().Address()`) są problematyczne. Dobre zasady tworzenia kodu narzucają, aby nie przekraczać dwóch poziomów połączenia (czyli np. `auto.Producer()`).

Generalnie można przyjąć zasadę, że im mniej wywołań procedur tworzonych obiektów i obiektów zwracanych przez nie, tym lepiej. Redukcja złożoności jest jedną z podstawowych przesłanek refaktoryzacji (McConnel, 2010).

Leniwa klasa

Leniwe klasy (Hunt, Thomas, 2000) to klasy, które niewiele robią lub mają nieprecyzyjnie zdefiniowane odpowiedzialności. Takie klasy często powstają w trakcie procesu refaktoryzacji, po reorganizacji w klasach pośrednich. Metody i dane z leniwych klas warto przenieść do innych klas, a z nich samych zrezygnować dla poprawy spójności kodu.

Jednocześnie klasy zawierające jedynie dane warto przekształcić tak, aby składowe stały się atrybutami innej klasy (klas). Podobnie klasy, dla których najlepszą nazwą byłby czasownik warto przekształcić w procedury innych klas (np. `DatabaseConnection`).

Niska efektywność

Leniwe i otyłe części kodu nie pozostają bez wpływu na efektywność. Przegląd kodu pod kątem optymalizacji wymaga profilowania, jednak warto zwrócić uwagę na typowe i dobrze znane operacje spowalniające program (McConnel, 2010):

- Operacje wejścia-wyjścia. Zbędne operacje plikowe, zbędne połączenia z bazą danych czy operacje sieciowe poważnie spowalniają działanie programu. W związku z tym, jeśli tylko jest to możliwe, warto wykonywać operacje w pamięci, szczególnie w miejscach, gdzie szybkość pracy ma duże znaczenie.
- Stronicowanie. Operacje wymagające wymiany stron pamięci działają wolnej od tych, które pracują na pojedynczych stronach.
- Wywołania systemowe. Warto minimalizować operacje wejścia-wyjścia z udziałem takich urządzeń jak dyski, drukarki, monitory czy klawiatura oraz inne operacje korzystające z zasobów systemowych.
- Błędy w kodzie. Pozostawienie kodu wspomagającego debugowanie, niepotrzebne odwołania do urządzeń czy bazy danych, zła organizacja obsługi błędów to typowe przykłady operacji zmniejszających wydajność.

Poprawki uwydatniają niespójności i zależność od innych klas

Wprowadzanie zmian jest dobrą okazją do przeprowadzenia dodatkowej refaktoryzacji. Zmiany mogą ujawnić istnienie ukrytych problemów.

Jeśli klasa ma więcej niż jedną podstawową odpowiedzialność, zmiany zwykle nie będą dotyczyły całej klasy, a jedynie jej części związanych z wybraną rolą. Jest to sygnał do sprawdzenia, czy nie lepiej byłoby podzielić klasę na mniejsze zgodnie z ich odpowiedzialnościami.

Z drugiej strony może się okazać, iż poprawki prowadzą do zmian w kilku różnych klasach, które jak się okazuje są ze sobą ściśle powiązane. Złączenia klas lub ich części mogą okazać się problematyczne, ale warto próbować je przeorganizować.

Publiczne składowe klasy

Rzadko się zdarza, aby istnienie publicznych składowych klasy było dobrym rozwiązaniem. O wiele bardziej elastycznym wyjściem jest dodanie do nich metod dostępowych.

Metody

Problemy związane z klasami przenoszą się często na metody. Istnieje wiele rodzajów błędów typowych dla metod. Większość z nich ma negatywny wpływ na czytelność i spójność, co w praktyce przekłada się na powstawanie trudnych do w ykrycia problemów. Błędy te wyszczególnione są poniżej.

Martwe metody

Metody, które nie są używane należy usuwać. Martwy kod (Hunt, Thomas, 2000) zaciemnia, niepotrzebnie komplikuje kod i może wprowadzić w błąd. Używanie systemu kontroli wersji pozwoli na przywrócenie treści metody jeśli zajdzie taka potrzeba.

Długa metoda

Długie metody to domena programowania strukturalnego. W projektowaniu obiektowym optymalna procedura cała powinna mieścić się na ekranie. Badania pokazują, że liczba wierszy metody nie powinna przekraczać 200 (Basili, Perricone, 1984). Zbyt długie metody wskazują na nieodpowiedni poziom modularności. Takie metody zwykle nie spełniają wymagania o jednej wyodrębnionej odpowiedzialności. Refaktoryzację takich metod najłatwiej przeprowadzić poprzez ich podział na mniejsze metody, z których każda ma zdefiniowane dokładnie jedno zadanie.

Długa lista parametrów

Problem długiej listy parametrów jest związany z brakiem kohezji pomiędzy modułami (klasami, metodami) aplikacji lub nieodpowiednim poziomem abstrakcji. Wiele małych, dobrze dopasowanych metod wymagających niewielu parametrów to lepsza modułowość i elastyczność aplikacji. W praktyce lista nie powinna zawierać więcej niż 7 parametrów (Miller, 1986).

Chaotyczna lub niewłaściwa lista parametrów

Parametry najlepiej układać jest w kolejności wejście – modyfikacja – wyjście (McConnel, 2010). Jest to najbardziej intuicyjny układ, który odpowiada sekwencji przeprowadzanych operacji. Jednak każda konsekwentnie stosowania kolejność jest ułatwieniem dla programisty. Pomocne może okazać się wprowadzenie konwencji

nazewniczych dla rodzaju parametrów na liście (np. z przedrostkami: `input_`, `modify_`, `output_`).

Warto też zawsze sprawdzać, czy przypadkiem do metody nie są przekazywane parametry, które nie są potem używane (Card i in, 1986).

Kolejnym elementem jest odpowiednie nazewnictwo i wykorzystanie parametrów. Przykładowo (Listing 5.13) wykorzystanie parametru wejściowego jako wartość zwracana funkcji jest przykładem nieprawidłowego podejścia.

Listing 5.13. Przykład nieprawidłowo wykorzystanych parametrów

```
int Function (int inputVal)
        inputVal=(inputVal-prevVal);
        inputVal=inputVal*multipleVal;
    return inputVal;
```

Parametr `inputVal` jest parametrem wejściowym. Jego zmiana i wykorzystanie jako wartość zwracana funkcji wprowadza niespójność. Lepiej by było wprowadzić dodatkową zmienną (Listing 5.14).

Listing 5.14. Poprawiony przykład z Listingu 5.13

```
int Function (int inputVal)
        ParamVal=(inputVal-prevVal);
        ParamVal =inputVal*multipleVal;
    return ParamVal;
```

Przykład źle napisanej metody

Listing 5.15. Przykład nieprawidłowo napisanej metody

```
void  ClientsProcess( Info & inputCost, int ff, ORD_DAT
ordRec, double estCost, double estSal, double bankSal, int
a,  int b, ORDER_st at  & newStat, ORDE  R_stat  & oldStat,
STATflag & fl, int Cltype)
    {
    int i;
    for (i=1; i<50; i++) {
       inputCost.cost[i]=0;
       inputCost.totalCost[i]=costExp[ff][i];
    }
    UpdateClient(ordRec);
    estSal=a*b*4.5/(double) ff;
    newStat= oldStat;
    fl=1;
    if (Cltype==1) {
      for(i=1;i<10;i++)
         profit[i]=Tprofit- estCost; } else
    if (Cltype==1)
      for(i=11;i<21;i++)
         profit[i]=Tprofit+ estCost; }
```

Metoda przedstawiona na Listingu 5.15 skupia bardzo wiele błędów wymienionych w tym rozdziale. Na liście błędów znajdują się:

- Zbyt długa lista parametrów metody, zawierająca dwanaście nieuporządkowanych parametrów.
- Zła nazwa metody, która nic nie mówi o wykonywanych akcjach.
- Brak dokumentacji kodu: komentarzy i samo-komentującego się kodu.
- Zła (lub nawet brak) organizacji logicznej kodu (różne style formatowania, różne konwencje wcięć, używania nawiasów).
- Zmienna wejściowa o nazwie `inputCost` jest modyfikowana, mimo jej nazwy wskazującej na to, że jest to parametr wejściowy.
- Wykorzystanie zmiennych globalnych.
- Różne poziomy abstrakcji.

- Brak jednego zdefiniowanego celu metody (części procedury nie są ze sobą związane).
- Brak zabezpieczenia przed niepoprawnymi operacjami (dzielenie przez 0).
- Istnieją magiczne liczby (4.5, 50, 10).
- Istnieją nieużywane parametry oraz parametr wyjściowy, który nie ma przypisanej wartości (parametr `oldStat`).

Zmienne

Zmienne są określonymi fragmentami pamięci posiadającymi nazwę i mogącymi przechowywać wartości uzależnione od typu zmiennej. Sposób ich wykorzystania powinien być zgodny z zasadami danego języka. Istnieje jednak kilka uniwersalnych rodzajów błędów, związanych z użyciem zmiennych.

Magiczne liczby

Magiczne liczby to literały liczb pojawiające się w środku kodu bez wyjaśnienia. Przykładem użycia magicznej liczby jest Listing 15.16.

Listing 5.16. Przykład użycia magicznych liczb

```
for i=1 to 50 do …
```

Magiczne liczby z Listingu 5.16 warto zmienić tak, jak to pokazuje Listing 15.17

Listing 5.17. Poprawiony przykład z Listingu 5.16

```
for i=1 to MAX_PAGE_NR do …
```

gdzie `MAX_PAGE_NR` jest zdefiniowaną stałą (lub w ostateczności zmienną globalną). Taka refaktoryzacja jest korzystna, ponieważ wymaga mniej pracy przy wprowadzaniu zmian i jest bardziej czytelny. Liczby takie jak 1 czy 0 używane do rozpoczynania pętli lub zmniejszania/zwiększania wartości powinny być jedynymi wyjątkami od zasady unikania magicznych liczb.

Źle dobrane typy danych

Warto zwracać uwagę na problemy, wynikające z niedopasowania typów. Dzielenie całkowite, przepełnienie w wynikach pośrednich, operacje dodawania/usuwania liczb zmiennoprzecinkowych znacznie różniących się rzędem wielkości, błędy zaokrągleń – to przykłady potencjalnych trudnych do wykrycia błędów.

Warto też używać zmiennych logicznych do upraszczania złożonych wyrażeń, takich jak na Listingu 5.18.

Listing 5.18. Przykład złożonego wyrażenia

```
If  (dokument.isOpen() And (Not docu   ment.EndOfFile())  And
(elementIndex<MAX_ELEMENT_NR) And (Not ErrorProcessing())
    …
End If
```

Wykorzystanie zmiennych logicznych (Listing 15.19) jest bardziej przejrzyste.

Listing 5.19. Poprawiony przykład z Listingu 5.18

```
documentReady =( dokument.isOpen() And
(Not document.EndOfFile() );
availableElement = (elementIndex<MAX_ELEMENT_NR);

If ((documentReady)  And (availableEle ment) And (Not
ErrorProcessing()) Then
    …
End If
```

Wśród typów danych warto wyróżnić typy wyliczeniowe. Ich użycie wspomaga niezawodność kodu oraz jest bardziej przejrzyste niż wykorzystywanie zmiennych logicznych (szczególnie jeśli opcji jest więcej niż dwie) lub magicznych liczb.

Problemy z deklaracją zmiennych

Warto sprawdzać (np. z wykorzystaniem kompilatora) czy w metodzie nie ma przypadkowego użycia dwóch podobnych nazw zmiennych lub też czy nie zostały

zadeklarowane zmienne, które nie są używane. Dodatkowo warto deklarować jawnie wszystkie zmienne, także jeśli nie wymaga tego kompilator. Deklarowanym zmiennym zawsze warto przypisać wartość podczas jej deklaracji lub w pobliżu jej pierwszego użycia. Pozwoli to na uniknięcie trudnych do zdiagnozowania błędów spowodowanych tym, że zmienna przyjmuje przypadkowe dane. Jeśli zmienne muszą być reinicjalizowane (np. liczniki w zagnieżdżonych pętlach), należy zawsze sprawdzić, czy inicjalizacja jest umieszczona w odpowiednim miejscu. Z kolei do inicjalizacji składowych klasy zawsze warto używać konstruktorów, co usprawni i ułatwi zarządzanie pamięcią.

Poniższe dwa przykłady (Listing 5.20 i Listing 5.21) prezentują dwa różne podejścia do deklarowania, inicjalizacji i korzystania ze zmiennych. Pierwszy z nich nie jest najlepszy, ponieważ taki logiczny podział (część deklaracji, inicjalizacji, użycia) nie redukuje możliwości powstania błędów i omyłkowych inicjalizacji, a zmienne sprawiają wrażenie, jakby miały być wykorzystywane w całej metodzie.

Listing 5.20. Przykład niepoprawnego podziału logicznego

```
int index;  //część deklaracji
double total;
boolean flag;
…
index=1; //część inicjalizacji
total=10.5;
flag=true;
…
if (flag) {   //część używająca zmiennych
…
```

Drugi przykład (Listing 15.21) prezentuje podział metody na części korzystające odpowiednio z poszczególnych zmiennych. Taka refaktoryzacja jest korzystna i ułatwia lepszą kontrolę kodu i podatność na zmiany.

Listing 5.21. Poprawiony przykład z Listingu 5.20

```
int index;
index=1;
…  //kod używający zmiennej index


double total;
total=10.5;
… //kod używający zmiennej total


boolean flag;
flag=true;
…  //kod używający zmiennej flag
if (flag) {
…
```

Deklarując zmienne warto pamiętać także o użyciu `final` czy `const`, co jest przydatne przy definiowaniu stałych klasy lub parametrów wejściowych. Chroni to przed niepożądanymi modyfikacjami.

Zmienne globalne

Obecność zmiennych globalnych utrudnia panowanie nad kodem i zmniejsza jego przejrzystość. Podczas refaktoryzacji zawsze warto zastanowić się, czy nie ma sposobu na zastąpienie ich innymi rozwiązaniami. Zmienne globalne nie posiadają mechanizmów ochrony dostępowej co przeczy założeniu hermetyzacji. Dodatkowo źle nazwane zmienne globalne mogą skomplikować zapis kodu i utrudnić tym samym poruszanie się po nim.

Omyłkowe zmiany wartości zmiennych globalnych są określane jako efekt uboczny i stanowią zwykle problem ukryty, dość trudny do zdiagnozowania. Innym problemem jest aliasowanie – spotykane, gdy zmienna globalna jest przekazywana w parametrze metody, która używa jej zarówno jako zmiennej globalnej jak i parametru metody.

Przykład aliasowania przedstawiono na Listingu 5.22.

Listing 5.22. Przykład aliasowania

```
void CurveFunction(int &  inputVar){
        inputVar =0;
        globalVal=inputVal*MAX_ITER;
    }
```

Wywołanie `CurveFunction(globalVal)` spowoduje omyłkową zmianę wartości zmiennej globalnej. Ten przykład uświadamia także, że używając zmiennych globalnych łatwo jest stracić pewność kolejności ich inicjalizowania i zmieniania.

Kolejnym problemem stwarzanym przez zmienne globalne jest zaburzenie modułowości i możliwości ponownego użycia kodu w innym miejscu czy innej aplikacji.

Dobrym podejściem minimalizacji błędów przy użyciu zmiennych globalnych jest wyposażenie ich w procedury dostępowe. Warto też stosować konwencję nazw dla zmiennych globalnych oraz starać się dobrze je opisać.

Zmienne o wielokrotnym przeznaczeniu

Każda zmienna powinna być używana tylko do jednego celu (Martin, 2010). Wykorzystanie jej w dwóch miejscach do dwóch różnych celów jest oczywiście możliwe, ale nie jest to dobra praktyka i taki kod wymaga refaktoryzacji. Nazwa zmiennej powinna być ściśle powiązana z jej zastosowaniem do jednego celu i dlatego zmienne do innych celów powinny być zdefiniowane oddzielnie. W szczególności należy się wystrzegać zmiennych „pomocniczych do wszystkiego" o nazwach `temp` czy `x`.

Kolejnym problemem jest używanie zmiennych o podwójnym, ukrytym znaczeniu (np. order nr oznacza numer zamówienia, a jej wartość równa -1 oznacza błąd). Takie podwójne role wprowadzają niejasności i brak przejrzystości kodu.

Obsesja typów podstawowych

Typy proste mogą być wykorzystywane do reprezentowania różnego typu obiektów. Jednak czasami warto, zamiast przeciążać proste typy danych, utworzyć małe klasy, które zapewnią lepszą kontrolę nad kodem. Wprowadzenie klas uchroni omyłkowe próby podstawienia np. temperatury pod prędkość, co możliwe by było w przypadku

przeciążenia typu `int` (Listing 5.23).

Listing 5.23. Wykorzystanie typu prostego do opisu zmiennej

```
List <int[]> markedCells = new ArrayList<int[]>();
for (int[] cell : theaterBoard)
…
```

Zamiast typu int, lepiej jest użyć prostej klasy zmiany (Listing 15.24).

Listing 5.24. Poprawiony przykład z Listingu 5.23

```
List <Cell> markedCells = new ArrayList<Cell>();
for (Cell cell : theaterBoard)
…
```

Warunki i pętle

Warunki i pętle są bardzo często wykorzystywanymi elementami programowania. Ich zrozumienie dla czytelnika będzie lepsze, jeśli programista zwróci uwagę na kilka uwag pozwalających uzyskać lepszą czytelność i przejrzystość. Warto też przejrzeć konstrukcje pod kątem wydajności – mniejsza liczba zagnieżdżeń pozytywnie wpływa na szybkość działania programu.

Warunki negatywne

Warunki negatywne są zwykle trudniejsze do zrozumienia, niż te pozytywne. Dlatego warto tak formułować instrukcje warunkowe, aby po słowie kluczowym `if` umieszczony był warunek pozytywny. Np. kod z Listingu 5.24 można zastąpić tym z Listingu 5.25.

Listing 5.24. Przykład wykorzystania warunku negatywnego

```
If ( user.isNotLogged() )
```

Listing 5.25. Przykład wykorzystania warunku pozytywnego

```
if ( !user.isNotLogged() )
```

Zbyt długa pętla

Jeśli istnieje możliwość wydzielenia fragmentów pętli do innych metod warto to zrobić, gdyż prowadzi to do redukcji złożoności pętli. Ogólnie długość pętli nie powinna przekraczać 50 wierszy a liczba jej poziomów zagnieżdżeń nie powinna być większa niż 3. Każda pętla powinna być spójna – dotyczyć tylko jednego zadania i być przejrzyście sformułowana. Z drugiej strony puste pętle są błędem i nie powinno się ich stosować. Istotne jest też zatrzymanie pętli, jeśli instrukcje w niej zawarte nie powinny być już wykonywane. Do tego celu należy zawsze używać instrukcji break zamiast znaczników logicznych lub zmiany indeksu pętli.

Zależności logiczne

Jeśli istnieje zależność pomiędzy dwoma modułami, to powinna być ona zależnością fizyczną, a nie jedynie logiczną. Nieprawidłowością jest, jeśli moduł zależny posiada założenia (zależności logiczne) na temat modułu głównego. W takiej sytuacji powinno się zastąpić założenie odpytaniem modułu głównego o pożądane informacje.

Struktura kodu

Struktura kodu ma newralgiczne znaczenie dla czytelności i przejrzystości treści programu. Niepoprawna struktura może też być przyczyną powstawania trudnych do wykrycia błędów.

Poszatkowanie

Każdy programista czytający kod doceni dobrą strukturę jego organizacji. Jeśli aby znaleźć szukane informacje trzeba prześledzić cały kod danej części to znak, że

część tą należy zrefaktoryzować pod kątem struktury. Warto zawsze grupować powiązane ze sobą jednym zadaniem instrukcje i oddzielać je od innych grup, wstawiając puste wiersze i/lub komentarz (Martin, 2010). Warto sprawdzać, czy grupy powiązanych instrukcji nie nachodzą na siebie. Poprzez taką refaktoryzację można znaleźć grupy tak niezwiązane z pozostałą częścią metody, że ich treść może zostać przeniesiona do nowej. Oczywiście poza podziałem kodu na grupy warto pamiętać o odpowiednich wcięciach kodu (choćby przy pętlach i instrukcjach warunkowych).

Przykładowy fragment metody (Listing 5.26) lepiej jest przedstawić w postaci takiej, jaką proponuje Listing 5.27.

Listing 5.26. Przykład poszatkowania

```
OrdersData ordersData;
SalesData salesData;
ClientsData clientsData;
ordersData.updateDaily();
salesData. updateDaily();
clientsData. updateDaily();
ordersData.generateReportMonthly();
salesData.generateReportMonthly();
clientsData.generateReportMonthly();
ordersData.print();
salesData.print();
clientsData.print();
```

Listing 5.27. Poprawiony przykład z Listingu 5.26

```
OrdersData ordersData;
ordersData.updateDaily();
ordersData.generateReportMonthly();
ordersData.print();

SalesData salesData;
salesData. updateDaily();
salesData.generateReportMonthly();
salesData.print();

ClientsData clientsData;
clientsData. updateDaily();
clientsData.generateReportMonthly();
clientsData.print();
```

Długi czas aktywności zmiennych

Związane z problem poszatkowania są także problem długiego czasu aktywności i duża rozpiętość. Czas aktywności zmiennej to liczba instrukcji części kodu, gdzie zmienna jest wykorzystywana. Pojęciem pokrewnym do czasu aktywności jest rozpiętość określająca liczbę instrukcji pomiędzy odwołaniami do zmiennej. Im mniejsze są wartości obu tych parametrów, tym lepiej. Zmniejsza się wtedy czas „potencjalnie niebezpieczny", którym zmiennej można omyłkowo zmienić wartość. Redukcja tych parametrów (tzw. Separacja pionowa) wpływa także pozytywnie na przejrzystość i czytelność kodu.

Układ kodu

Dobry układ kodu wskazuje logiczną strukturę programu, znacznie zwiększa jego czytelność i pomaga przy kolejnych refaktoryzacjach i zmianach.

Istotnym elementem układu jest organizacja blokowa grup instrukcji, w szczególności warunków i pętli. Granice bloków określają zwykle pary `begin-end` lub nawiasy klamrowe. Zawartość bloku sformatowana powinna być z wykorzystaniem wcięć. Wcięcia nie powinny być jednak zbyt głębokie, gdyż zwykle utrudnia to czytanie kodu (Wingerd, Seiwald, 2008, s. 545).

Listing 5.28 to przykład kodu z głębokimi podwójnymi wcięciami, brakiem nawiasów (które nie są w tym przykładzie wymagane ze względu na pojedyncze instrukcje w każdej z opcji).

Listing 5.28. Przykład kodu z głębokimi wcięciami

```
if (productCount>20 && productPrice>5)
    if (productCount>80 && productPrice>15)
      if(productCount>150)
                        discount=20;
                  else
                        discount=10;
                              else
                                    discount=5;
                        else
                              discount=2;
```

W większości przypadków o wiele lepiej jest wykorzystać mniejsze wcięcia. Dodatkowo warto rozpoczynać i kończyć blok nawiasami lub parami `begin-end`, bez których kod jest mniej czytelny. Listing 5.29 prezentuje kod, w którym, usunięte są podwójne wcięcia, których użycie narusza w rzeczywistości zasadę uzyskania logicznego podziału.

Dodatkowymi elementami, na które warto zwrócić uwagę to:

- widoczność zakończenia instrukcji;
- niewstawianie więcej niż jednej instrukcji w wierszu;
- wstawianie komentarzy równo z sąsiadującym kodem;
- poprzedzanie komentarzy przynajmniej jednym pustym wierszem;
- rozdzielanie sąsiadujących procedur;
- zapisywanie każdej klasy w osobnym pliku;
- jednolitość stylu formatowania.

Dobra organizacja kodu ułatwia czytelność i przyspiesza pracę. Stosowanie tych samych zasad dla całej organizacji dodatkowo wspomaga pracę programistów pracujących przy kilku projektach.

Listing 5.29. Poprawiony przykład z Listingu 5.28

```
if (productCount>20 && productPrice>5)
{
    if (productCount>80 && productPrice>15)
   {
       if(productCount>150)
         {
               discount=20;
         }
         else
         {
               discount=10;
         }
   }
   else
   {
         discount=5;
   }
}
else
{
   discount=2;
}
```

Komentarze

Większość twórców oprogramowania stosuje komentarze w kodzie. Często zdarza się jednak, że komentarze są bezużyteczne, wprowadzają w błąd lub nie wnoszą nic nowego do kodu aplikacji. Znacznie lepiej jest polegać na samokomentującym się kodzie uzupełnionym jednie o komentarze zawierające dodatkowe, przydatne informacje. Istnieje kilka typów błędów, na które warto zwrócić uwagę.

Nadmiarowe komentarze

Źle napisane komentarze znacznie utrudniają czytanie kodu. Język naturalny nie jest tak precyzyjny jak kod, dokładny opis wymaga też większej ilości słów. Komentarze nie powinny ponawiać i objaśniać kodu – za to powinna być odpowiedzialna właściwa struktura i jakość kodu (Fowler i in, 2000). Komentarze powinny informować o celu, jaki ma kod oraz być swoistym jego spisem treści lub podsumowaniem ułatwiającym

znalezienie poszukiwanego fragmentu. Jednocześnie poziom abstrakcji komentarzy powinien być wyższy niż poziom abstrakcji kodu.

Nie ma więc żadnego uzasadnienia dla umieszczenie np. oczywistego komentarza z Listingu 5.30.

Listing 5.30. Przykład nadmiarowego komentarza

```
    // funkcja zwraca dzień miesiąca
    //@return dzień miesiąca
public int getDayOfMonth(){
      return DayOfMonth;
    }
```

Dobrze jest dodawać 1-2 zdania komentarza do akapitów/bloków kodu umieszczając je przed blokiem (w szczególności dotyczy to pętli i instrukcji warunkowych). Komentarz taki powinien określić cel bloku kodu, a nie być jego opisem, powtarzającym kod. Podobnie 1-2 zdania komentarza umieszczonego przed procedurą wystarcza zwykle do jej opisania. Dla klas przydatne jest umieszczenie na początku opisu sposobu jej projektowania.

Pliki również powinny być opatrzone odpowiednimi komentarzami. Na początku pliku powinno być umieszczona informacja o przeznaczeniu pliku i jego zawartości, w tym o jego klasach i procedurach. Warto też dodać dane kontaktowe autora, znacznik wersji.

Przestarzałe komentarze

Warto przejrzeć komentarze po wprowadzeniu zmian – komentarze mogą łatwo się dezaktywować, co jest później źródłem niejasności. Złe komentarze są gorsze niż kod pozbawiony komentarzy. Istotne znaczenie ma też styl pisania komentarzy. Jeśli jest on zbyt pracochłonny, zmiany i konserwacja kodu będą wymagały większej pracy przy modyfikacji komentarzy, co zwiększa prawdopodobieństwo niepoprawnych zmian w komentarzach lub odłożenia ich na później.

Przykład z Listingu 5.31 ilustruje komentarze, które są estetycznie ułożone, ale nie wspomagają zmian, które wymagałyby wiele czasu poświęconego na formatowanie.

Dodatkowe rzędy kropek, ułożenie gwiazdek, układ tekstu po przejściu do kolejnych linii jest uciążliwy i niepodatny na zmiany.

Listing 5.31. Przykład komentarza trudnego w utrzymaniu

```
/************************************
*                                   *
* klasa Enemy                       *
*                                   *
* autor: M.Nowak                    *
* data: 12.08.2012                  *
*                                   *
* Procedury sterujące ruchem postaci  *
* w grze. Punktem wejścia jest      *
* procedura ManageEnemy()           *
…
// Zmienna          Znaczenie
//-------          --------
//xPos.........współrzędna X
//yPos.........współrzędna Y
//needReload....wymaga przeładowania (=0 gdy
//nie wymaga,
//                                  (=1 gdy wymaga)
…
```

Istotne jest także przeglądanie kodu pod kątem znaczników, które wskazują np. na niedokończony kod lub kod do poprawy. Aby uniknąć przykrych niespodzianek pod koniec projektu warto używać standaryzowania, co ułatwi wyszukiwanie znaczników w kodzie.

Zakomentowany kod

Pozostawianie zakomentowanego kodu to jedna z najgorszych praktyk programistycznych. Autor tworząc kod powinien po jego przetestowaniu powinien go oczyścić z wszelkich pomocniczych, ale niepotrzebnych już fragmentów, w szczególności fragmentów niepotrzebnego kodu. Zostawianie zakomnetowanego kodu sprawia wrażenie, że jest on do czegoś potrzebny, dlatego inni programiści zwykle nie usuwają takiego kodu, z który z czasem zalega, zanieczyszcza kod i jest trudny do zrozumienia.

Komentarze do pojedynczych wierszy

W dobrym kodzie opisywanie pojedynczych wierszy jest rzadkością (Martin, 2010). Używa się ich jeśli wiersz jest na tyle skomplikowany, że wymaga komentarza (być może w takich sytuacjach warto poprawić kod) lub wiersz zawierał błąd, o którym programista chce zapisać informację. Komentarzy na końcu wiersza można użyć do opisywania deklaracji danych (w tym ich zakładany zakres i zakodowane wartości) lub zakończenia bloków.

Warto unikać komentarzy na końcu wiersza, ponieważ zaciemniają układ kodu, wydłużają wiersze i trudno się je formatuje. Ograniczona ilość miejsca sprawia też, że pisane są one skrótowo i często są mało zrozumiałe. Jeśli komentarz ma dotyczyć grupy wierszy, nie powinno się go dodawać na końcu jednego z nich (włączając w to pętle i instrukcje warunkowe).

Spaghetti code

Programista powinien dążyć do uzyskania samo-komentującego się kodu. Dokumentacja na poziomie kodu to nic innego jak poprawnie napisany kod, który nie wymaga nawet komentarzy. Jego podstawą jest prawidłowy styl programowania, na który składa się poprawna struktura i układ, dobre nazwy, unikanie zmiennych globalnych i literałów, stosowanie stałych, redukcja złożoności, zachowanie odpowiedniego poziomu abstrakcji. Przeciwieństwem tego sposobu programowania jest tzw. Spaghetti code, którego mianem określa się kod skomplikowany, trudny do zrozumienia.

Komentarze objaśniające trudniejsze fragmenty kodu

W takich przypadkach zamiast poświęcać czas na dokładny opis kodu o wiele lepiej będzie przepisać go tak, aby był łatwiejszy do zrozumienia dla czytelnika.

Nazewnictwo

Błędy i niedociągnięcia w nazewnictwie wydają się nieznaczące podczas pisania kodu. Jednak ich znaczenie wzrasta podczas ponownej analizy kodu, kiedy niejednoznaczne i niejasne sformułowania utrudniają czytanie programu i jego modyfikację. Refaktoryzacja pod kątem nazewnictwa powinna uwzględniać kilka

poniżej wyjaśnionych typów błędów.

Niejednoznaczność

To najczęściej spotykany i jednocześnie najpoważniejszy problem związany z nazewnictwem. Nazwa zmiennej czy klasy powinna być czytelna, łatwa do zapamiętania i przede wszystkim wskazująca na przeznaczenie zmiennej lub klasy. Aspekt ten określić można jako ukierunkowanie na problem. Nazwa zmiennej np. powinna udzielać odpowiedzi na pytanie ‘co’, a nie np. ‘jak’ lub ‘dlaczego’. Nazwa procedury powinna zawierać opis działania lub zwracanej wartości, często może to być np. połączenie czasownika z przedmiotem operacji.

Przykłady prawidłowych nieprawidłowych nazw zmiennych prezentuje Tabela 5.1, a nazw klas – Tabela 5.2.

Tabela 5.1. Przykłady nazw zmiennych

Przeznaczenie zmiennej	Dobra nazwa	Zła nazwa
Suma zamówień klienta	clientOrdersSum, clientOrdersTotal	client, total, CLOT, order, x, a1, SumOfTotaldSingleClientOrder
Bieżąca data	currentDate, todayDate	DT, dat, dd, actual, date, curd, c, TheDateOfToday
Numer wiersza	lineNumber	line, linno, Nr, l, x1, NumerOfActualLine
Wartość podatku	taxRate	tax, TT, taxes, a, b, ValueOfTaxValue
Powierzchnia mieszkania	apartmentArea, placeArea, flatArea	area, apartment, Flat, ar, totalArea, AreaOfApartmentInBuilding
Flaga określająca typ raportu	reportType	flag, reportFlag, repFF

Tabela 5.2. Przykłady nazw klas

Przeznaczenie klasy	Dobra nazwa	Zła nazwa
Pobierz dane klienta	ReadClient, GetClient	client, getC, ClientDB
Znajdź książkę po Id	GetBookById	GetBooksById, GetBook, book
Zarządzanie rachunkiem	AccountManager	Account, A_dir, Managing

Istotnym, często popełnianym błędem jest nieodpowiednie nazywanie zmiennych stanu. Warto wprowadzić inne nazwy niż popularna flag, która tak naprawdę niewiele

mówi o rodzaju przechowywanej informacji. Lepiej jest używać przyrostka `Type` lub określić zupełnie inną nazwę. Nie jest też dobrą praktyką używanie magicznych liczb dla określenia wartości znacznika. Jeśli jest to możliwe, powinno się je zastąpić wartościami `true/false` lub wcześniej zdefiniowanymi i nazwanymi typami (np. `reportType= ReportType_Monthly`).

Nieodpowiednia długość

Zbyt krótkie nazwy niosą ze sobą za mało informacji o zmiennej czy klasie a ich charakter wydaje się być tymczasowy, pomocniczy. Z drugiej strony nadmiernie rozciągnięte nazwy mogą zaciemniać kod, utrudniać jego czytanie i przede wszystkim zwiększają podatność kodu na literówki. Złotym środkiem jest znalezienie opisującej nazwy, której długość nie przekroczy 16 znaków.

Istnieją też poprawne nazwy, które z założenia powinny być krótkie, ponieważ są jedynie zmiennymi roboczymi o krótkim czasie widoczności. Do takich zmiennych należą na przykład liczniki pętli (zwykle oznaczane jako i,j) – o ile nie są wykorzystywane do dalszych obliczeń poza pętlą.

Brak konwencji nazewnictwa

Konwencja nazewnictwa powinna być zdefiniowana przed rozpoczęciem pracy z kodem. Istnieje kilka dobrych praktyk, które warto wdrożyć:

- Używanie nazw angielskich zamiast polskich.
- Oddzielenie nazwy zmiennej od nazwy klasy (np. rozpoczynanie nazwy zmiennej małą literą, a kolejne człony nazwy pisane dużą, bez znaku '_'; rozpoczynanie nazwy klasy dużą literą); wyróżnianie stałych.
- Wyróżnianie zmiennych globalnych.
- Używanie określonych kwalifikatorów dla zmiennych przechowujących wyniki prostych operacji algebraicznych (`Total, Sum, Max, Min, Average, Total`) – kwalifikatory te powinny być dołączane na końcu nazwy.
- Używanie typowych nazw dla wartości logicznych (`done, error, found , success/ok`) lub użycie ich jako przyrostku (np. `connectionOk, isFound`); nie należy natomiast używać przeczeń (np. `notFound`)

- Określone przedrostki dla typów wyliczeniowych wskazujących, że należą one do jednej grupy (np. `ReportType_Monthly`, `ReportType_Daily`, `ReportType_Annual`).

Dodatkowo warto unikać w nazwach następujących zwrotów utrudniających pracę z kodem:

- skróty, które ciężko wymówić;
- połączenia niejasne, mogące prowadzić do nieporozumień;
- nazwy, które ciężko wymówić;
- nazwy o podobnym znaczeniu;
- liczby w nazwach (np. `file1` i `file2`);
- nazwy różniące się tylko wielkością liter;
- nazwy standardowych typów, zmiennych, metod (nawet jeśli kompilator na to zezwala).

Stosowanie wyżej wymienionych reguł i wypracowanie własnej konwencji nazewniczej ułatwi oraz przyspieszy pracę zarówno z własnym kodem jak i kodem innych programistów stosujących tą samą notację.

Wiele języków w jednym skrypcie

Dzisiejsze środowiska programistyczne umożliwiają umieszczanie fragmentów w wielu różnych językach w jednym pliku. W szczególności dotyczy to aplikacji internetowych. Często trudno jest tego uniknąć, warto dążyć do tego, aby pojedynczy skrypt zawierał jak najmniej wymieszanych fragmentów kodu napisanych w różnych językach. O wiele lepiej jest umieszczać kod różnych języków w oddzielnych skryptach.

Nieprawidłowe działanie w warunkach granicznych

Pisząc i refaktoryzując kod należy przewidzieć i sprawdzić możliwie największą ilość przypadków granicznych, które mogą wystąpić sporadycznie podczas działania aplikacji. Należy zabezpieczać działanie aplikacji przed wystąpieniem błędów wynikających z braku lub nieodpowiedniej obsługi problemów, warunków granicznych i wyjątków. Skutecznym sposobem na radzenie sobie z warunkami granicznymi jest tworzenie testów.

5.3. METODY REFAKTORYZACJI

Podstawową zasadą projektowania i tworzenia kodu jest **identyfikacja potencjalnych zmian**. Jest to jedna z podstawowych praktyk wspomagająca refaktoryzację i jednocześnie jedno z trudniejszych programistycznych działań. Identyfikacja obszarów potencjalnych zmian umożliwia izolację takich obszarów w taki sposób, aby skutki zmian miały jak najmniejszy zasięg. W szczególności warto zwrócić uwagę na takie aspekty jak reguły biznesowe (zmiany organizacyjne, prawne, itd), zależność od sprzętu, mechanizmy wejścia-wyjścia, nietypowe lub tymczasowe rozwiązania związane z językiem programowania.

Cały proces obsługi obszarów potencjalnych zmian składa się z trzech etapów:

- Identyfikacja elementów, po których można oczekiwać zmian. Lista takich elementów powinna zawierać informacje o potencjalnych modyfikacjach.
- Oddzielenie elementów, po których można oczekiwać zmian. Najlepszym sposobem na od dzielenie takich elementów jest umieszczenie ich w osobnych klasach lub zgrupowanie ich w klasach przechowujących elementy o podobnie niestabilnym charakterze.
- Izolacja elementów, po których można oczekiwać zmian. Izolacji dokonuje się poprzez odpowiednią konstrukcję interfejsów w taki sposób, aby zminimalizować ich podatność na skutki zmian. Zmiany te powinny pozostać wewnątrz klas, a rolą interfejsu jest zapobieganie zakłóceniu pracy innych klas z nich korzystających.

Metod i zasad refaktoryzacji jest wiele i mogą być definiowane na różnych poziomach abstrakcji. Wiele z nich zostało opisanych w Podrozdziale 5.2 opisującym typowe błędy w kodzie wraz z proponowanymi zasadami ich naprawy. Wybrane metody refaktoryzacji przedstawiają możliwe przekształcenia na różnych poziomach (McConell, 2010): danych, instrukcji, metod, klas i ich składowych oraz całego systemu. Do istotnych elementów refaktoryzacji można też zaliczyć przygotowania do procesu refaktoryzacji, dokumentowanie pracy oraz stosowanie wzorców projektowych.

Przygotowywanie kodu na zmiany jest elementem idei **programowania defensywnego**. Jego podstawowym celem jest takie zabezpieczenie kodu, aby zmiany lub błędne wywołania w innej części programu nie spowodowały szkód. Do

metod stosowanych w programowaniu defensywnym należą:

- Zabezpieczenia przed nieprawidłowymi i niespodziewanymi danymi wejściowymi (źródła zewnętrzne, parametry wejściowe do procedur, wartości nullowe, wprowadzanie zasady obsługi błędnych danych).
- Asercje (procedury (zwykle predykaty), które funkcjonują jako mechanizm automatycznej kontroli; zazwyczaj przyjmują formę wyrażenia logicznego, które, jeśli zwraca prawdę, oznacza poprawne działanie kodu) – np. `assert underTheRoot <0 : „Wyrażenie pod pierwiastkiem jest ujemne”;`
- Korzystanie z mechanizmów obsługi błędów i wyjątków.
- Stosowanie kodu wspomagającego debugowanie (i usuwanie go, gdy nie jest już potrzebny).

Poniżej przedstawione zostały szczegółowo poszczególne rodzaje refaktoryzacji.

Refaktoryzacja na poziomie danych to metody, które poprawią organizację danych i usprawnią korzystanie ze zmiennych. Należą do nich:

- Zastąpienie magicznych liczb stałymi.
- Zmiana niejasnych nazw na bardziej opisowe (o wiele lepiej jest zmienić nieintuicyjną nazwę niż opisywać ją dodatkowymi komentarzami).
- Wprowadzanie zmiennych tymczasowych o opisowych nazwach przechowujących wyniki częściowe wyrażeń.
- Zastąpienie wyrażenia procedurą (co zwykle jest związane z eliminacją powtórzeń kodu).
- Eliminacja zmiennych wykorzystywanych do wielu celów poprzez wprowadzenie wielu zmiennych o sprecyzowanych nazwach.
- Zastąpienie operacji na parametrach wejściowych procedury operacjami na zmiennych lokalnych. Zmiana wartości parametrów wejściowych procedury może być myląca.
- Zastąpienie danych prostych klasą, szczególne jeśli dane proste wymagają uzupełnienia innymi danymi lub zachowaniami. Operowanie na obiektach zwiększa poziom hermetyzacji i daje lepszą kontrolę.
- Przekształcenie zbioru niepowiązanych kodów w typ wyliczeniowy lub klasę. Ułatwia to jego użycie, rozbudowę i obsługę błędów.

- Przekształcenie tablicy zawierającej różnego rodzaju typy w obiekt.
- Przekształcenie rekordów w obiekt.

Refaktoryzacja na poziomie instrukcji to metody, które poprawią budowę instrukcji programu:

- Dekompozycja wyrażeń logicznych. Warto wprowadzać dobrze nazwane zmienne pomocnicze, aby ułatwić zrozumienie skomplikowanych wyrażeń.
- Wprowadzanie funkcji logicznych aby zmniejszyć poziom złożoności wyrażeń. Dodatkowo wprowadzenie funkcji logicznych eliminuje powtórzenia i ułatwia wprowadzanie zmian.
- Przeniesienie z bloków instrukcji warunkowych elementów powtarzających się. Jeśli te same fragmenty kodu umieszczone są po `if` i po `else`, warto przenieść je poza blok.
- Zastąpienie zmiennej sterującej pętli konstrukcją `break` lub `return`.
- Wykorzystanie instrukcji return zamiast przypisywania gotowej wartości zmiennej w instrukcjach warunkowych. Czytelność kodu wzrośnie, jeśli wyjście z bloku nastąpi od razu po określeniu zwracanej wartości.
- Zastąpienie powtarzających się instrukcji warunkowych `case` polimorfizmem. Wykorzystanie hierarchii dziedziczenia może uczynić kod bardziej czytelnym i ustrukturyzowanym.
- Używanie stworzonych przez siebie obiektów pustych zamiast wartości null.

Refaktoryzacja na poziomie metod to metody, które poprawią konstrukcję poszczególnych metod:

- Wyodrębnianie oddzielnej metody, jeśli w metodzie istnieje kod powtarzający się lub działający na innym poziomie abstrakcji.
- Połączenie metod o podobnych działaniach.
- Podzielenie długiej procedury na mniejsze, ewentualnie połączenie ich w osobną klasę.
- Upraszczanie skomplikowanych algorytmów.
- Redukcja zbyt dużej liczby parametrów metody.

- Oddzielenie metod pobierających od modyfikujących.
- Podzielenie metody, której działanie uzależnione jest od przekazanego parametru (parametrów)
- Przekazywanie do metody całych obiektów zamiast pojedynczych parametrów.

Refaktoryzacja na poziomie implementacji klasy to metody, które poprawią konstrukcję klas:

- Analiza i ewentualna zmiana hierarchii dziedziczenia (przenoszenie procedur, pól, zawartości konstruktora). Takie działanie wspomaga eliminację powtórzeń i ułatwia uzyskanie specjalizacji klas.
- Wyodrębnienie podklasy z klasy, której obiekty wykorzystują tylko elementy kodu.
- Połączenie podobnego kodu dwóch klas i przeniesienie go do nadklasy.

Refaktoryzacja na poziomie interfejsu klasy to metody, które pozwalają na ulepszenie interfejsów klas:

- Przeniesienie procedury lub jej treści do innej klasy, jeśli jej poziom abstrakcji lub odpowiedzialność jest bardziej odpowiednia.
- Połączenie leniwej klasy z inną o podobnej odpowiedzialności.
- Połączenie leniwą metodę z inną, aby ich praca była bardziej zrozumiała.
- Podzielenie klasy na mniejsze w przypadku, gdy można w niej wyodrębnić kilka obszarów odpowiedzialności.
- Ukrywanie lub przeniesienie delegacji klas.
- Usuwanie zbędnych pośredników pomiędzy klasami.
- Dodanie klasy rozszerzającej, jeśli danej klasy nie można bezpośrednio modyfikować.
- Zmiana widzialności publicznych danych klasy na prywatne i utworzenie metod dostępowych.
- Utworzenie nowego interfejsu klasy w przypadku, gdy używane są tylko niektóre metody klasy. Nowy interfejs powinien udostępniać tylko te potrzebne metody.

Refaktoryzacja na poziomie systemu to metody, które pozwalają na ulepszenie całego systemu:

- Przechowywanie danych w pamięci podręcznej. Często pobierane i rzadko aktualizowane dane warto przetrzymywać w pamięci podręcznej i traktować je jako główne źródło danych.
- Zmiana kierunkowości relacji pomiędzy klasami z jednokierunkowej na dwukierunkową lub na odwrót, w zależności od tego, czy obie powiązane klasy muszą korzystać nawzajem ze swoich funkcji, lub też nie.
- Ujednolicenie sposobów obsługi błędów w projekcie.

Jedną z metod wspomagających refaktoryzację jest też **dokumentowanie pracy**. Formalne podeście polegające na tworzeniu ustrukturyzowanych dokumentów jest tylko jedną z alternatyw. Szczególnie w przypadku mniejszych projektów dobrze sprawdzają się również inne rozwiązania lub ich kombinacja. Należą do nich:

- Dołączanie dokumentacji do kodu. Dodawanie dobrej jakości podsumowujących komentarzy oraz komentarzy zawierających ważne decyzje konstrukcyjne w nagłówkach plików lub klas. Tworzenie samo-komentującego się kodu.
- Utrwalanie ważniejszych decyzji, schematów oraz ciekawszych rozwiązań w bazie wiedzy (np. w systemie zarządzania wiedzą, wiki, itp.). Ułatwi to śledzenie przebiegu pracy nad projektem i utrwali ciekawe pomysły, które mogą przydać się w przyszłości.
- Używanie kart CRC (ang. *Class-Responsibility-Collaboration*, pol. klasa-odpowiedzialność-współpraca), z których każda opisuje jedną klasę. Na karcie zapisuje się jej nazwę, zakres odpowiedzialności i powiązania z innymi klasami. Taki nieformalny sposób opisu może zostać zebrany w formę dokumentacji.

Podczas refaktoryzacji warto przestrzegać kilku ogólnych zasad:

- Zachowywanie początkowej wersji kodu, aby w razie potrzeby można było ją przywrócić (korzystanie z systemu kontroli wersji).
- Minimalizowanie zasięgu refaktoryzacji, przeprowadzanie refaktoryzacji pojedynczo sprawdzając ich wpływ na pozostałe elementy kodu.
- Tworzenie listy przekształceń i listy „przekształceń oczekujących”.

- Korzystanie z testów jednostkowych i tworzenie nowych testów dla nowych fragmentów kodu.
- Nienadużywanie refaktoryzacji - wprowadzanie poprawek do niedziałającego kodu lub zmiana założeń nie może być określane mianem refaktoryzacji.
- Refaktoryzację warto jest przeprowadzać iteracyjnie na każdym z etapów projektu.
- Refaktoryzacja źle napisanego kodu nie ma sensu – taki kod lepiej jest przepisać.

Refaktoryzację warto jest przeprowadzać podczas tworzenia nowej metody, klasy lub podczas usuwania błędów. W szczególności warto zwracać uwagę na moduły złożone lub/i podatne na błędy.

Refaktoryzację należy przeprowadzać często, gdyż nie można doprowadzić do sytuacji takiej, w której refaktoryzacji wymaga duży moduł aplikacji. Jest to zbyt ryzykowne i może doprowadzić do powstania niespodziewanych błędów.

5.4. NARZĘDZIA REFAKTORYZACJI

Dzisiejsze narzędzia wspomagające refaktoryzację są najczęściej wbudowane w środowiska programistyczne (IDE) (Fowler i in, 2000; McConnel, 2010). Kontrolują one składnię i uzupełniają działania kompilatora. Ułatwiają też przegląd kodu dzięki wyświetlaniu skróconych listingów (tzw. "zwijane" fragmenty) oraz łatwemu przechodzeniu do w ybranych definicji i wywołań klas, procedur, zmiennych a także możliwości podglądu hierarchii klas (tzw. opisu drzew dziedziczenia). Kontrolują składnię i uzupełniają działania kompilatora. Ułatwiają też przegląd kodu dzięki wyświetlaniu skróconych listingów (tzw. "zwijane" lub „ukrywane" fragmenty kodu) oraz łatwemu przechodzeniu do wybranych definicji i wywołań klas, procedur, zmiennych, a także możliwości podglądu hierarchii klas (tzw. opisu drzew dziedziczenia).

Środowisko ułatwia też poprawne formatowanie kodu i organizację wcięć. Nadają też spójny wygląd komentarzom, generują strony HTML dla każdej metody. Narzędzia takie wspomagają też ujednolicenie kodu pochodzącego z różnych źródeł. Przydatne są także generatory opisu interfejsu, generujące szczegółowy opis interfejsu na

podstawie zawartości plików źródłowych. Przykładowym narzędziem wspomagającym generowanie opisu interfejsu jest Javadoc.

Innym typem narzędzi są też wielopoziomowe mechanizmy cofania operacji i wyszukiwanie z wykorzystaniem wyrażeń regularnych. Integracja z systemami kontroli wersji ułatwia przywracanie poprzedniej wersji kodu po nieudanej próbie refaktoryzacji. Narzędzia takie wspomagają kontrolę kodu oraz zarządzanie kolejnymi wersjami oprogramowania wraz z opisem wymagań oraz testów. Innym narzędziem jest porównywanie plików z wyświetlaniem różnic w sąsiadujących oknach. Przydatne jest ono do wycofywania niepotrzebnych poprawek, ułatwia pracę zespołową nad kodem. Jednocześnie eliminuje potrzebę zostawiania zakomentowanych fragmentów kodu w programie. Do tej samej grupy należą także narzędzia wspomagające scalanie, które umożliwiają pracę kilku osób nad tym samym plikiem. Wspomagają one rozwiązywanie konfliktów ze zmianami innych członków zespołu.

Istnieją także narzędzia służące do analizy kodu pod katem jego jakości. Narzędzia takie generują raporty dotyczące złożoności metod i klas, określają statystyki dotyczące liczby wierszy, zmiennych, ilości komentarzy a także liczbie i jakości zmian w projekcie.

Warto także zwrócić uwagę na narzędzia (wbudowanych zwykle w IDE) usprawniające zmianę nazw klas, atrybutów lub metod. Wspomagają one także zmianę liczby lub kolejności parametrów metod. Istnieją także narzędzia wspomagające przekształcanie struktury kodu, np. konwersję instrukcji goto.

Istotną rolę w procesie tworzenia i refaktoryzacji kodu odgrywają także narzędzia do debugowania i testowania. Te pierwsze zawierają ostrzeżenia kompilatora, profilowanie, narzędzia porównujące wersje plików źródłowych, monitory diagnostyczne). Narzędzia testujące to systemy automatycznego testowania (JUnit, NUnit), generatory zautomatyzowanych testów, narzędzia zakłócające pracę systemu, wprowadzające i monitorujące defekty.

Do przykładowych narzędzi dedykowanych wspomaganiu refaktoryzacji można zaliczyć:

- JustCode – dodatek do Microsoft Visual Studio .NET. Usprawnia analizę kodu i sprawdza błędy. Ułatwia nawigację i wyszukiwanie. Posiada moduły generacji oraz formatowania kodu, wspiera także stosowanie szablonów.

- ReSharper - dodatek do Microsoft Visual Studio .NET. Posiada analizę statyczną kodu (wyświetlanie błędów, ostrzeżeń, sugestii, podpowiedzi), automatyczną korekcję błędów, optymalizator, wyszukiwanie wielofunkcyjne oraz nawigację. Wspomaga poprawę struktury plików, szybką nawigację, tworzenie klas, zmiennych oraz ułatwia konstrukcję dziedziczenia. Wspiera też tworzenie testów jednostkowych.
- CodeRush – kolejny dodatek do Microsoft Visual Studio .NET. Analiza statyczna kodu umożliwia korekcję błędów, uzupełnianie i generowanie kodu, podświetlanie składni, formatowanie. Dodatkowo posiada kilkadziesiąt wbudowanych refaktoryzacji (obsługa zmian nazw, metody dostępowe, zmiana kolejności atrybutów, zmiana typów danych, enkapsulacja, upraszczanie wyrażeń warunkowych, itd.). Wspomaga także testy jednostkowe (integracja z takimi narzędziami jak NUnit, XUnit, MSTest).
- DMS Software Reengineering Toolkit – zestaw narzędzi wspierających analizę i modyfikację systemów. Wspomaga wiele języków programowania. Obejmuje analizę kodu (typy danych, zmienne lokalne i globalne, zasięg, komentarze, formatowanie), wspomaganie zmian, zarządzanie gramatykami bezkontekstowymi, obsługę błędów, konstrukcję schematów przepływu danych.
- Xcode – IDE dedykowane dla rozwoju aplikacji pod Mac OS X. Wspiera rozwój oprogramowania i jego zmiany, rozwój dokumentacji, zarządzanie błędami.
- RefactorIt – narzędzie wspomagające refaktoryzację, zawierające metryki kodu źródłowego, wspieranie audytów i korekcję błędów, generator kodu.
- VisualAssist – narzędzie dedykowanie dla języka C++. Wspomaga sprawdzanie pisowni i nawigację w projekcie, wyświetla podpowiedzi i posiada sporo opcji konfiguracyjnych.

5.5. PYTANIA KONTROLNE

1. Wyjaśnij cele i ideę refaktoryzacji.
2. Jak często należy przeprowadzać refaktoryzację?
3. Scharakteryzuj najbardziej popularne błędy dotyczące kodu metod klas i podaj sposoby ich usunięcia.
4. Wymień typowe błędy komentowania.
5. Scharakteryzuj metodę refaktoryzacji na poziomie danych.

Literatura

Ackerman F., Buchwald L., Lewski F. (1989), *Software inspections: an effective verification process*, IEEE Software, s.31-36.

Agile Project Management (2011), *CC Pace Systems,* dostępny 16 sierpnia 2011, <http://www.ccpace.com/Resources/documents/AgileProjectManagement.pdf>

Alexander Ch. (1977), *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press.

Ambler S. W. (2011), *Best Practices for Software Development*, dostępny 16 sierpnia 2011, < http://www.ambysoft.com/essays/agileLifecycle.html>

Basili V., Selby R. (1987), *Experimentation in Software Engineering*, IEEE Transactions on Software Engineering, SE-12, s.733-743.

Basili V.R., Perricone B.T. (1984), *Software errors and complexity: an empirical investigation*, Communications of the ACM, 27, 1, s. 42-52.

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

Bender J., MCWerter J. (2011), *Professional Test-Driven Development with C#*, Wiley Publishing, USA.

Bijay K., Patton C. (2007), *Oprogramowanie godne zaufania*, Gliwice, Helion.

Bruegge B., Dutoit A. H. (2011), *Inżynieria oprogramowania w ujęciu obiektowym: UML, wzorce projektowe i Java*, Helion.

Card D.N., Church V.E., Agresti W.W. (1986), *An Empirical study of software design practices*, IEEE Transactions on Software Engineering, SE-12, 2, s. 845-851.

Chaos Summary 2009 (2009), Standish Group, Boston.

Coad P., Lefebvre P. i De Luca P. (1999), *Java Modeling In Color With UML: Enterprise Components and Process*, Prentice Hall.

Cockburn A. (2004), *Crystal Clear: A Human-Powered Methodology for Small Teams*, Addison-Wesley.

Cockburn A. (2008), *Agile Software Development. Gra zespołowa*, Gliwice, Helion.

Cohn M. (2005), *Agile Estimating and Planning*, Prentice Hall PTR.

Cohn M. (2011a), *Don't average during planning poker*, dostępny 27 sierpnia 2011, <http://blog.mountaingoatsoftware.com/dont-average-during-planning-poker>

Cohn M. (2011b), *When should we estimate the Product Backlog*, dostępny 27 sierpnia 2011, <http://blog.mountaingoatsoftware.com/when-should-we-estimate-the-product-backlog>

Cooper J. W. (2001), *Java. Wzorce projektowe*, Helion.

Czarnacka-Chrobot B. (2004*), Z najnowszych „Kronik Chaosu" Standish Group, czyli czy uczymy się na błędach?* w Grabara J., Nowak J. (red) *Efektywność zastosowań systemów informatycznych*, Warszawa, WNT, s. 209-238.

Dajda J. (2008), *Supporting agile methodologies in distributed setting*, rozprawa doktorska, AGH, Kraków.

Dikstra E. (1968), *Go To statement considered harmful*, Communications of the ACM, 11, 3, s. 147-148.

Elssamadisy A. (2010), *Agile. Wzorce wdrażania praktyk zwinnych*, Gliwice, Helion.

Flasiński M. (2008), *Zarządzanie projektami informatycznymi*, Warszawa, PWN.

Florac W., Carleton A. (1999), *Measuring The Software Process*. Addison-Wesley, Boston.

Fowler M. (2011), *Development of Further Patterns of Enterprise Application Architecture*, dostępny 20 sierpnia 2011, <http://www.martinfowler.com/eaaDev/>

Fowler M., Beck K., Brant J., Opdyke W., Roberts D. (2000), *Refactoring: Improving the Design of Existing Code*, Nowy Jork, Addison-Wesley.

Foy C. (2011), *Story Points Are Dead! (Long Live Story Points?)*, dostępny dnia 27 sierpnia 2011, <http://blog.coryfoy.com/2011/04/story-points-are-dead-long-live-story-points/>

Freeman E., Freeman E., Sierra K., Bates B. (2005), *Head First Design Patterns*, Gliwice, Helion.

Gamma E., Helm R., Johnson R., Vlissides J. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional.

Grand M. (1998), *Patterns in Java – Volume 1*, John Wiley & Sons, Inc., NY.

Greene J., Stellman A. (2011), *Wideband Delphi estimation process*, dostępny 27 sierpnia 2011, < http://www.stellman-greene.com/aspm/content/view/23/38/>

Gurses L. (2006), *10 Mistakes In Transitioning to Agile. Slow down the transition in order to go f ast*, , dostępny 16 sierpnia 2011, Dr. Dobb's Portal, <http://www.ddj.com>

Haugen N. (2011), *Planning poker*, dostępny dnia 27 sierpnia 2011, <http://www.infoq.com/interviews/nils-haugen-planning-poker>

Highsmith J. (2004), *Agile Project Managment: Creating Innovative Products*, Addison-Wesley Professional.

Hunt A., Thomas D. (2000), *The Pragmatic Programmer: From Journeyman to Master*, Nowy Jork, Addison-Wesley.

Jasiński M. (2006), *Wymagania pozafunkcyjne i ISO 9126*, Materiały na platformie <http://wazniak.mimuw.edu.pl/>

Jayaswal B., Patton P. (2009), *Oprogramowanie godne zaufania*, Helion, Gliwice.

Johnson M. (1994), *Dr. Boris Beizeron software testing*, The Software QA Quarterly, s.7-13.

Jones C. (1986), *Programming productivity*, McGraw-Hill, Nowy Jork.

Jones C. (1996), *Software defect-removal efficiency*, IEEE Computer.

Jones C. (2000), *Software assessments, benchmarks and best practices*, Addison-Wesley, Reading, MA, USA.

Kircher M., Prashant J. (2006), *Zarządzanie zasobami. Wzorce projektowe*, Gliwice, Helion.

Koszlajda A. (2010), *Zarządzanie projektami IT. Przewodnik po metodykach*, Gliwice, Helion.

Krebs J. (2009), *Agile Portfolio Management*, Redmond, Microsoft Press.

Martin R, Martin M. (2008), *Agile. Programowanie zwinne*. Helion, Gliwice.

Martin R. (2010), *Czysty kod*, Gliwice, Helion.

Martin R. C. (2011), *Design Principles and Design Patterns*, dostępny 16 sierpnia 2011, <http://www.objectmentor.com/resources/articles/Principles_and_Patterns>

Martin R., Martin M. (2008), *Agile. Programowanie zwinne: zasady, wzorce i praktyki zwinnego wytwarzania oprogramowania w C#*, Gliwice, Helion.

McConnell S. (2006), *Software Estimation*, Microsoft Press, Washington, USA.

McConnell S. (2010), *Kod doskonały. Jak tworzyć oprogramowanie pozbawione błędów*, Helion, Gliwice.

Meyer B., Nawrocki J., Walter B. (Eds.) (2007), *Balancing Agility and Formalism in Software Engineering*. Lecture Notes in Computer Science 5082, Springer.

Miller G.A. (1986), *The magical number seven, plus or minus two: some limits on our capacity for processing information*, The Psychological Review, 63, 2, pp. 81-97.

Miłosz M. (2006), *Zarządzanie projektami informatycznymi – pomiędzy formalizmem a elastycznością*, w Miłosz M., Grabara J.K (red.), *Dylematy zarządzania projektami*, Katowice, PTI, s. 9-22.

Miłosz M. (red.) (2003), *Przedsięwzięcia wdrożeniowe – od teorii do praktyki*, Warszawa, MIKOM.

Miłosz M., Borys M. (2008), *Szacowanie projektów internetowych – metoda Story Points*, w *Aplikacje internetowe – od teorii do praktyki*, PTI, Warszawa, s. 141 166

Mountain Goat Software (2011), *Planning Poker*, dostępny dnia 27 sierpnia 2011, <http://planningpoker.com/>

Myers G.J. (2005), *Sztuka testowania oprogramowania*, Helion, Gliwice.

Nawrocki J. (2006), *Model dojrzałości CMMI*, Materiały na platformie <http://wazniak.mimuw.edu.pl/>

Nawrocki J. (2006), *Normy serii ISO 9000*, Materiały na platformie <http://wazniak.mimuw.edu.pl/>

Nawrocki J., Janiszewski T., Jasiński M. (2001), *Wprowadzenie do Programowania Ekstremalnego*, materiały VIII Konferencji Systemy Czasu Rzeczywistego, Wrocław.

Palmer S., Felsing S. (2002), *A Practical Guide to Feature-Driven Development*, Prentice Hall.

Paulk M. (2001), *Extreme Programming from a CMM Perspective*, IEEE Software, Vol. 18 (2001), No 6, s.19-26.

Savoia A. (2008), *Piękne testy*, w Oram A., Wilson G. (red.), *Piękny kod*, Gliwice, Helion, s.105-123.

Schwaber K. (2005), *Sprawne zarządzanie projektami metodą Scrum*, Warszawa, APN PROMISE.

Schwaber K. (2007),*The Enterprise and Scrum*, Microsoft Press.

Shore J., Warden S. (2008), *Agile Development. Filozofia programowania zwinnego*, Gliwice, Helion.

Shull I in. (2002), *What we have learned about fighting defects*. Metrics 2002, IEEE, s.249-258.

Skroban K. (2010), *Testowanie oprogramowania*, Zawiła-Niedźwiecki J., Rostek K., Gąsiorkiewicz K. (red.), *Informatyka gospodarcza 2*, Warszawa, C.H. Beck, s. 205-232.

Source Making, dostępny 16 sierpnia 2011, <http://sourcemaking.com/design_patterns/proxy>

State of Agile Survey (2010), Versione, 8 s. (http://www.versionone.com/pdf/2010_State_of_Agile_Development_Survey_Results.pdf)

Szyjewski Z. (2004), *Metodyki zarządzania projektami*, Wydawnictwo Placet, Warszawa.

Thompson B. (2011), *Better estimates with Wideband Delphi*, dostępny dnia 27 sierpnia 2011, <http://leansoftwareengineering.com/wideband-delphi/>

Wiegers K. (2002), *Peer reviews in software: A practical guide*, Addison-Wesley, Boston.

Wiegers K.E. (2007), *Practical Project Initiation. A Handbook with Tools*, Microsoft Press, Washington.

Wingerd L., Seiwald C. (2008), *Kod w ruchu*, w Oram A., Wilson G. (red.), *Piękny kod*, Gliwice, Helion, s.545-555.

Wrycza S., Marcinkowski B., Wyrzykowski K. (2006), *Język UML 2.0 w modelowaniu systemów informatycznych*, Helion.

Yourdon E. (2000), *Marsz ku klęsce. Poradnik dla projektanta systemów*, Warszawa, WNT.

Indeks