1 Oprogramowanie

Oprogramowanie jest pojęciem podstawowym w inżynierii oprogramowanie. Poza kodem źródłowym oraz binariami, w skład oprogramowania w pojęciu inżynierii oprogramowania wchodzą także dokumentacja, pliki konfiguracyjne itp.

2 Inżynieria oprogramowania

Inżynieria oprogramowania zajmuje się wytwarzaniem oprogramowania we wszystkich fazach jego cyklu życia. Inżynieria rozważa w kontekście praktycznym rozmaite aspekty wytwarzania oprogramowania. Zajmuje się także dostarczaniem metod i standardów do modelowania, a później tworzenia dobrego oprogramowania, przy czym dobre oprogramowanie w pojęciu IO – punkt 3.

3 Cechy dobrego oprogramowania

- poprawność, zgodność z wymaganiami użytkowników
- łatwość konserwacji i dokonywania zmian
- niezawodność (availability, reliability), bezpieczeństwo (safety, security)
- wydajność, efektywne wykorzystanie zasobów
- łatwość stosowania, ergonomia

4 Podstawe czynności

- określanie wymagań i specyfikacja wymagań
- projektowanie i implementacja
- testowanie – walidacja i weryfikacja
- konserwacja

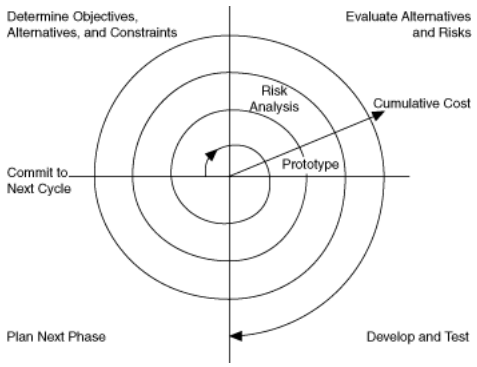
5 Modele cyklu życia

5.1 Model spiralny

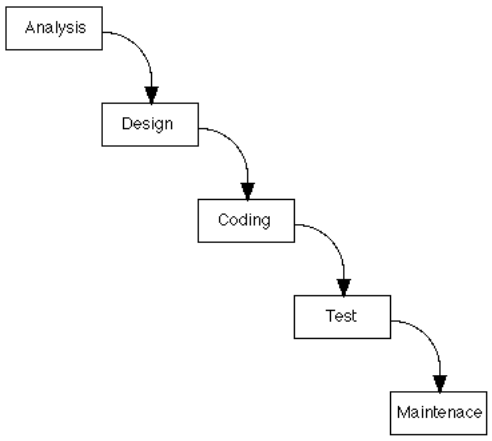
Model zaproponowany jako ogólny schemat rozwoju systemów i oprogramowania. Wyróżnia się cztery podstawowe etapy: określanie celu, analiza alternatyw i ryzyka, realizacja i walidacja, planowanie następnego etapu. **Wady**: słabo ujmuje specyfikę wytwarzania oprogramowania, **Zalety**: jawne uwzględnienie alternatyw i ryzyka

5.2 Model kaskadowy

Model, w którym każda kolejna faza następuje dopiero po zakończeniu fazy poprzedniej. W klasycznym modelu kaskadowym (*waterfall model*) nie ma powrotu do poprzedniej fazy. **Zalety**: zidentyfikowanie podstawowych faz i uporządkowanie procesu tworzenia oprogramowania, **Wady**: rygorystyczne określenie następstwa faz, co może utrudniać realizację projektu.



Rysunek 1: Model spiralny



Rysunek 2: Model kaskadowy

5.3 Model ewolucyjny

Model, którego celem jest poprawienie modelu kaskadowego przez rezygnację ze ścisłego, liniowego na następstwa faz. Pozostawia się te same czynności, z możliwością powrotu do poprzednich faz. Tym samym umożliwia się adaptację zmian i korekcję błędów. Model wymaga dodatkowych strategii dla uporządkowania procesu

5.3.1 Prototypowanie

Technika w ramach modelu ewolucyjnego, w której pojawia się nowa faza – tworzenie prototypu. **Prototyp** jest niepełnym systemem, spełniającym część wymagań, przeznaczonym do testowania rozwiązań. Z założenia prototyp nie wchodzi w skład ostatecznego systemu. Prototyp posiada swoją specyfikę: służą do wykrystalizowania i ostatecznego ustalenia wymagań, do stworzenia prototypu także należy użyć modelu.

5.3.2 Wytwarzanie odkrywcze

Wytwarzanie odkrywcze jt. model ewolucyjny, w którym możliwość powrotu dotyczy całego cyklu wytwarzania oprogramowania. Istotą jest stała współpraca z klientem, który otrzymuje kolejne, coraz bogatsze wersje systemu i na ich podstawie uszczegóławia swoje wymagania. Model dobrze radzi sobie z częstymi zmianami wymagań klientów. Konieczna jest strategia zarządzania wersjami.

5.4 Model komponentowy

W modelu komponentowym po fazie określania wymagań następuje faza analizy możliwości wykorzystania istniejących, gotowych komponentów i ew. faza modyfikacji wymagań w konsekwencji stosowania komponentów. W fazie projektowania uwzględnia się już znalezione komponenty oraz ew. nowe, związane z implementacją. Projekt wykonywany jest tak, aby te spośród wytwarzanych elementów, które się do tego nadają, mogły być ponownie wykorzystane jako komponenty. W fazie tworzenia kodu zwraca się szczególną uwagę na interfejsy pomiędzy komponentami. **Wady**: wymagania narzucone przez komponenty mogą być niezgodne z wymaganiami klientów, modyfikacja kodu może być utrudniona przez brak kontroli nad zewnętrznymi komponentami.

6 Narzędzia CASE

CASE – Computer Aided Software Engineering, Inżynieria Oprogramowania Wspomagana Komputerowo.

- niskiego poziomu – edytory i kompilatory (GCC, VI etc.)
- wspomagające poszczególne fazy – analizy, projektowania, implementacji, testowania – Visio, BoUML
- zintegrowane środowiska wytwarzania (integrated development environments, IDE) – Visual Studio, Oracle JDeveloper

7 Fazy procesu określania wymagań

Określanie wymagań może odbywać się na różnych etapach realizacji projektu:

- *feasibility study* – w fazie przeprowadzania studium wykonalności projektu
- po podjęciu decyzji o realizacji ale przed podpisaniem kontraktu
- po podpisaniu kontraktu

Na każdym z etapów inny jest zakres wymagań, poziom szczeółowości opisu itp. Proces określania wymagań można podzielić na następujące fazy:

- ustalania wymagań
- specyfikacji wymagań
- walidacji wymagań

Fazy te mogą być powtarzane wielokrotnie na różnych etapach określania wymagań, wraz z rosnącym zakresem i poziomem szczegółowości wymagań i proponowanych modeli. Klient musi ustalić swoje wymagania w kontekście możliwości i ograniczeń, wykonawca musi dostosować funkcje programu do standardów i konwencji dziedziny zastosowań.

8 Wymagania funkcjonalne i niefunkcjonalne

9 Rodzaje wymagań niefunkcjonalnych

- funkcjonalne – wszystkie funkcje i operacje, które ma realizować system. Powinny być kompletne (opisywać wszystkie funkcje) i spójne (nie zawierać stwierdzeń sprzecznych)

- niefunkcjonalne – mówią, w jaki sposób mają być realizowane funkcje; zasoby, ograniczenia czasowe, niezawodność (MTBF, prawdopodobieństwo wystąpienia awarii), bezpieczeństwo, przenośność, normy, standardy, przepisy itd.

Niekiedy wymagania nf na pewnym etapie szczegółowości, mogą stać się wymaganiami funkcjonalnymi na innym etapie szczegółowości. Zarządzanie wymaganiami jest konieczne, gdyż w praktyce wymagania ciągle się zmieniają. Oznacza proces kontroli zmian wymagań i ich konsekwencji dla systemu w ciągu całego cyklu życia. Istotnym elementem jest znalezienie powiązań pomiędzy: wymaganiami a motywacjami, wymaganiami między sobą, wymaganiami, a elementami projektu i implementacji będącymi konsekwencjami wymagań.

10 Techniki stosowane przy odkrywaniu wymagań

- poznanie całości otoczenia systemu
- wykorzystanie istniejących systemów realizujących podobne funkcje
- obserwacje i wywiady z przyszłymi użytkownikami
- stosowanie scenariuszy wykorzystania systemu
- modelowanie systemu
- tworzenie prototypów
- rozważanie punktów widzenia (viewpoints)

11 Punkty widzenia

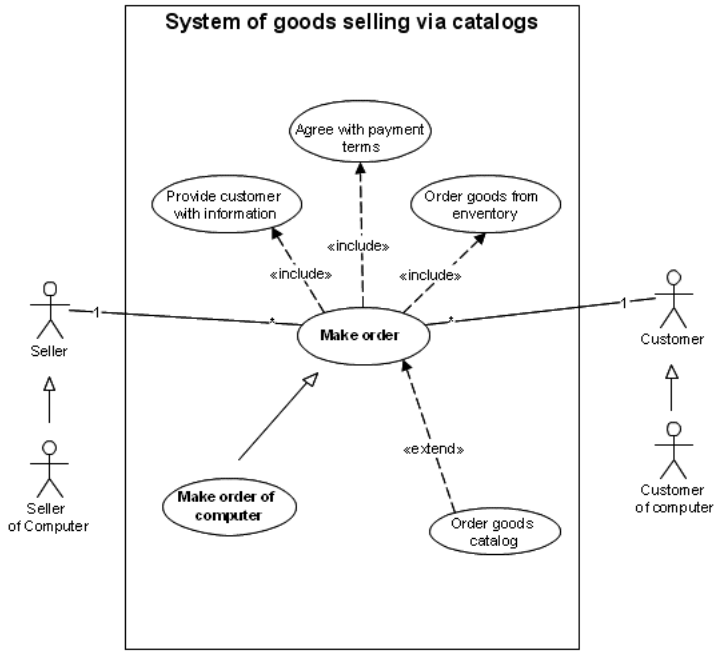
Punkt widzenia określa dowolną osobę, której dotyczy funkcjonowanie systemu lub element otoczenia mający wpływ na system. Punkty widzenia dzielimy na:

- bezpośrednie – związane z ludźmi bezpośrednio korzystającymi z systemu
- pośrednie – związane z ludźmi pośrednio zainteresowanych funkcjonowaniem systemu
- związane z dziedziną – standardy, przepisy itp.

Aktor w UML reprezentuje osobę lub element otoczenia wchodzący w interakcję z systemem i odgrywający pewną rolę. Reprezentuje punkt widzenia na modelu UML.

12 Przypadki użycia a scenariusze

Scenariusz jt. opis możliwych sekwencji zadań związanych z systemem, przypadek użycia jest zbiorem realizacji funkcji lub grup funkcj i oznacza interakcję z całym systemem lub podsystemem prowadzącą do konkretnego rezultatu. Dla określenia wymagań istotne znaczenie mają przypadki nietypowe (awarie, błędy itp.)



Rysunek 3: Diagram przypadku użycia

13 Use case diagram

14 Scenariusz

```
Nazwa: Dokonaj rezerwacji
Inicjujący: Rezerwujący
Cel: zarezerwować pokój w hotelu
Główny scenariusz:
1. Rezerwujący zgłasza chęć dokonania rezerwacji
2. Rezerwujący wybiera hotel, datę, typ pokoju
3. System podaje cenę
4. Rezerwujący prosi o rezerwację
5. Rezerwujący podaje swoje dane
6. System dokonuje rezerwacji i nadaje jej unikalne ID
7. System przesyła rezerwującemu ID rezerwacji
Rozszerzenia:
3. Pokój niedostępny
   a. System podaje inne możliwości wyboru
   b. Rezerwujący dokonuje wyboru
```

15 Specyfikacja metody

```
Nazwa: obliczPrędkośćWentylatora
Rola: Oblicza prędkość tak, aby utrzymać założoną temperaturę
Opis: Na podstawie odczytu aktualnej temperatury wewnątrz i na zewnątrz
oblicza prędkość wentylatora
Dane wejściowe: brak argumentów bezpośrednich, aktualne odczyty z czujników,
poprzednie odczyty
Źródła danych: czujniki: wywołanie procedury (...), pamięć lokalna:
zapamiętane poprzednie odczyty
```

```
Elementy współpracujące: (...)
Warunki początkowe: poprawność aktualnych oraz poprzednich odczytów
Dane wyjściowe: prędkość obrotowa wentylatora w RPM
Efekty uboczne: zmiana poprzednich odczytów
Warunki końcowe: prędkość kątowa obliczona poprawnie. Poprzedni odczyt
na podstawie aktualnego odczytu.
```

16 Formalna specyfikacja wymagań

17 Ostateczny opis wymagań

Standardy tworzenia: US, DoD, IEEE

- przedmowa (docelowi czytelnicy, historia wersji)
- wstęp
- słownik pojęć
- definicja wymagań użytkowników
- ogólna architektura systemu
- specyfikacja wymagań systemowych
- modele systemu
- opis ewolucji system
- dodatki
- indeks

18 Walidacja wymagań

Walidacja określa, czy uzyskane ostatecznie wymagania odpowiadają pierwotnym oczekiwaniom użytkownika. Sposoby: przegląd wymagań, tworzenie prototypów, tworzenie planów testów.

19 Modele systemu

Modele systemu dzielimy na dwie podstawowe grupy:

- modele zachowania – aspekt dynamiczny. Działania należą zwykle do dwóch grup: zmiana stanu systemu oraz przetwarzanie danych wejściowych w dane wyjściowe.
- modele struktury – aspekt statyczny. Ukazują statyczne elementy systemu i ich wzajemne zależności.

20 Sposoby użycia UML w procesie developmentu

Z teoretycznego punktu widzenia UML może być stosowany jako sposób uściślenia pojmowania bytów występujących w programach. Z praktycznego punktu widzenia może być stosowany jako notacja ułatwiająca zrozumienie systemu. W procesie wytwarzania oprogramowania UML może być stosowany na kilka sposobów:

- jako pomocnicza notacja do zrozumienia struktury i funkcjonowania
- jako sposób zapisu szczegółowego projektu systemu
- jako ścisły opis tworzonego programu, aby możliwa była automatyczna generacja programu

21 Diagramy i perspektywy spojrzenia w UML

UML 2.0 wyróżnia trzynaście rodzajów diagramów w hierarchii:

- struktury: diagramy klas, obiektów, komponentów, struktur złożonych, pakietów, wdrożenia
- zachowania: przypadków użycia, maszyny stanowej, czynności, interakcji: przeglądu interakcji, sekwencji, komunikacji, czasowe

Autorzy UML rozróżniają pięć perspektyw spojrzenia na system informatyczny:

- przypadków użycia: diagramy use case, interakcji, stanów i czynności
- projektowa: diagramy klas, obiektów, interakcji, stanów i czynności
- procesowa: diagramy j/w ze szczególnym uwzględnieniem klas i obiektów aktywnych
- implementacyjna: diagramy komponentów, interakcji, stanów i czynności
- wdrożeniowa: diagramy wdrożenia, interakcji, stanów i czynności

22 Forward i reverse engineering

Inżynieria wprzód (ang. Forward engineering) to proces w którym najpierw następuje dokładne modelowanie i projektowanie systemu, a dopiero następnie jego implementacja. Tworzenie kodu odbywa się na podstawie możliwie kompletnych modeli.

Inżynieria odwrotna (ang. reverse engineering) to proces badania produktu (urządzenia, programu komputerowego) w celu ustalenia jak dokładnie działa, a także w jaki sposób i jakim kosztem został wykonany. Zazwyczaj prowadzony w celu zdobycia informacji niezbędnych do skonstruowania odpowiednika lub prezentacji działania. Innym zastosowaniem jest porównanie lub zapewnienie współdziałania z własnymi produktami. Przykład: Samba – serwer udostępniania plików i folderów protokołu Microsoft SMB dla Linuksa.

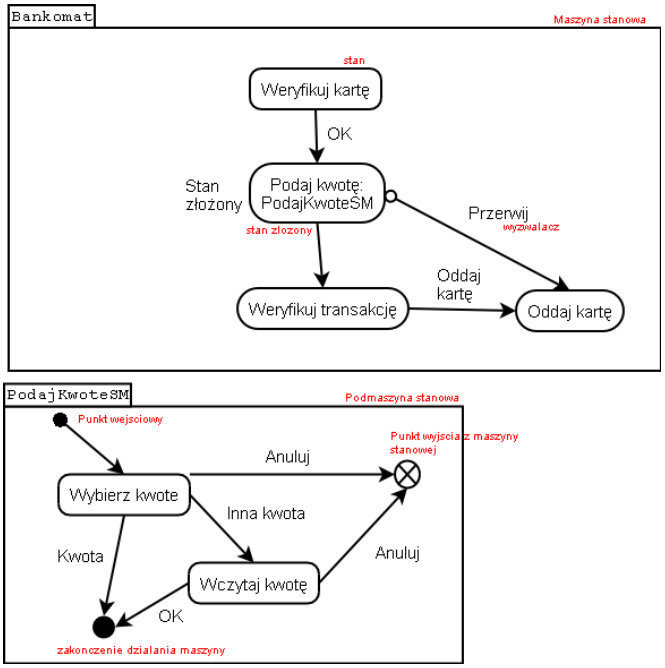
23 Przydatność diagramów stanu

Modele maszyn stanowych są to modele ukierunkowane na zmiany stanów systemu, szczególnie przydatne w systemach czasu rzeczywistego (RTS) a także systemach sterowania. W przypadku OOP maszyna stanowa może dotyczyć obiektu. Diagram stanu dla obiektu obrazuje jego cykl życia. UML 2.0 wprowadza rozróżnienie pomiędzy:

- maszynami stanu modelującymi zachowanie (behavioral state machines) – tworzą model zachowania danego systemu (np. bankomatu)

- maszynami stanu reprezentującymi dowolne sposoby użycia klasyfikatorów (protocol state machines)

24 Diagram maszyny stanowej zachowania

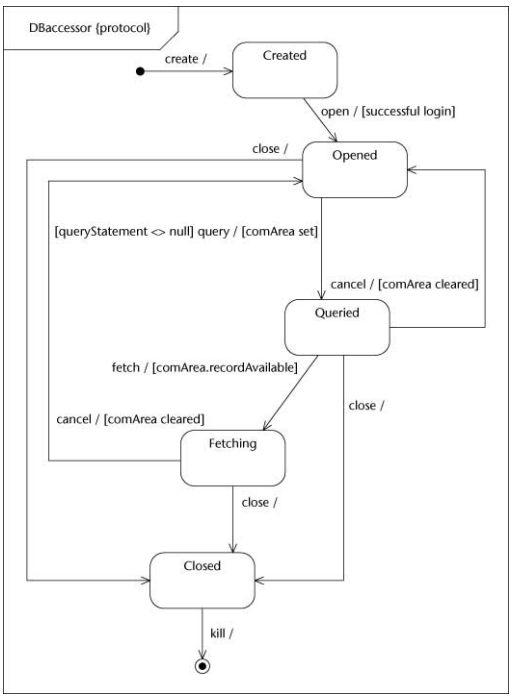


Rysunek 4: Diagram maszyny stanowej zachowania

25 Diagram maszyny stanowej protokołu

26 Metodologia strukturalna

Metody strukturalne powstały w latach ’70. Dominuje sposób analizy *top-down* czyli od ogółu, do szczegółu. Podstawowym sposobem podejścia jest **dekompozycja funkcjonalna** – podstawowa funkcja systemu jest rozbijana na sekwencję fuknicji składowych realizujących kolejne etapy przetwarzania. Proces dekompozycji aż do znalezienia funkcji nadających się do bezpośredniej implementacji. **Zaleta**: stosunkowa prostowa i szybkość realizacji; **Wady**: silne uzależnienie elementów modelu od siebie, niska elastyczność, słabe uwzględnienie ponownego wykorzystania kodu, trudności w testowaniu (system nie nadaje się do testowania, dopóki wszystkie elementy nie są gotowe); **Przydatność**: szybkie, wstępne stworzenie modelu, małe i średnie proste systemy, tworzenie algorytmów



Rysunek 5: Diagram maszyny stanowej protokołu

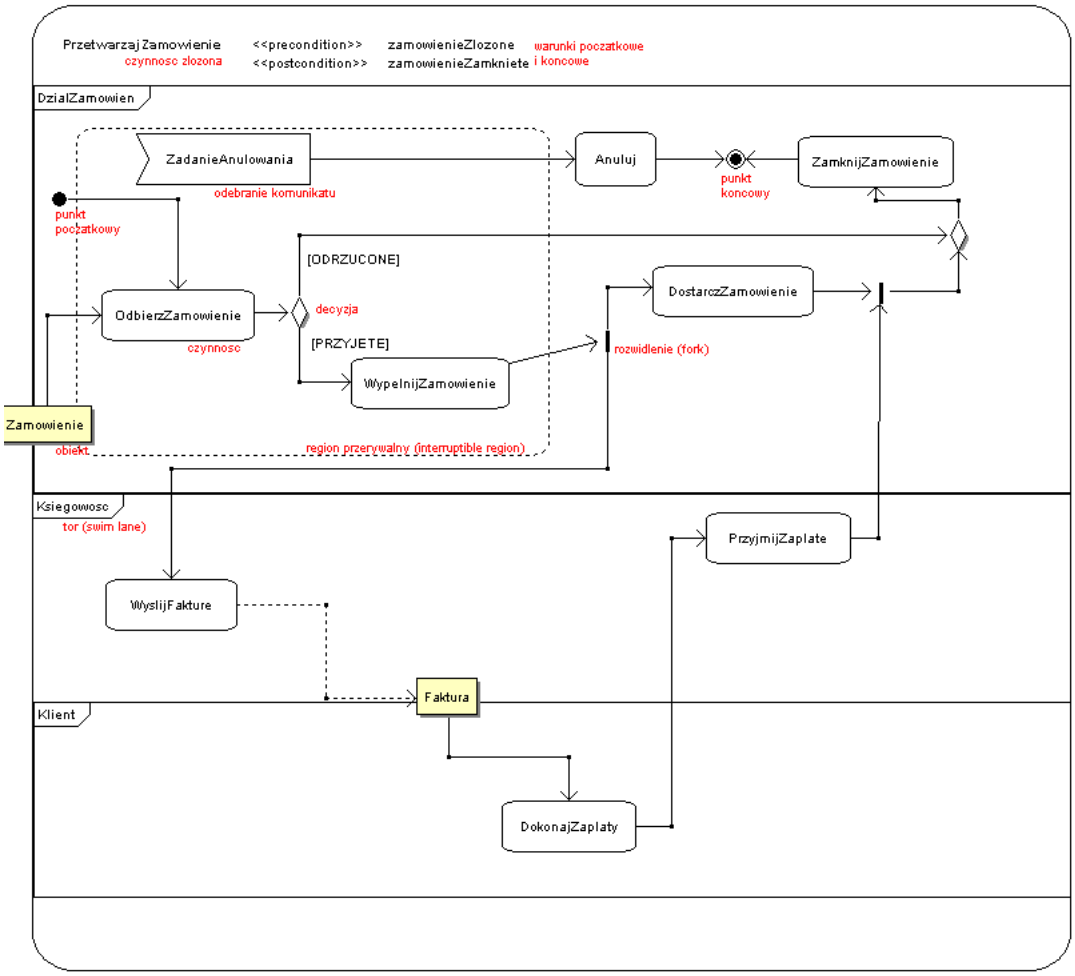
27 Diagram czynności UML

28 Węzły sterowania na diagramie czynności

- decyzje – opisują ścieżki alternatywne; do wyboru jednej z nich dochodzi na podstawie wyliczonych wartości warunków (wyrażeń logicznych) lub else - reprezentującego ścieżkę wybieraną, gdy wszystkie inne warunki nie są spełnione
- rozwidlenia (forks) – opisują współbieżne wykonanie czynności; w punkcie scalenia dochodzi do synchronizacji współbieżnych przepływów sterowania; Wątek warunkowy - jeśli warunek jest fałszywy, zakłada się, że z punktu widzenia scalenia wątek ten jest juz zakończony
- oczekiwanie na sygnał – oznacza oczekiwanie określony okres czasu na sygnał. Jeśli sygnał nie nadejdzie, wykonywana jest procedura anulowania czynności określona na diagramie zgłoszenie wyjątku

29 Elementy strukturalne oprogramowania

- komponent — fragment oprogramowania nadający się do niezależnego montowania w większe programy. Komponent nadaje się bezpośrednio do użycia – jest tworzony i kompilowany niezależnie od reszty programu. Mają precyzyjnie zdefiniowany interfejs określający usługi. Postrzegane od strony klienta.
- podsystem – część programu wydzielona ze względu na funkcjonalność oraz niezależność działania składników moduł
- klasa – byt precyzyjnie definiowany w programowaniu orientowanym obiektowo. Postrzegany jako element systemu od strony programisty
- pakiet – zbiór klas. Podobnie jak klasa, jest elementem systemu postrzeganym od strony programisty, nie klienta. rozszerzenie (extension)



Rysunek 6: Diagram czynności UML

30 Modele i architektury

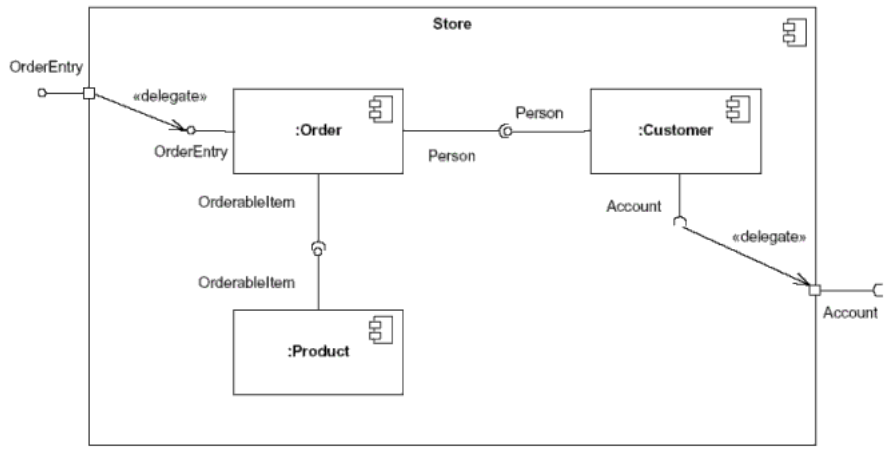
Modele sterowania:

- centralny: wywołań i powrotów, zarządcy i pracowników (głównie w przetwarzaniu współbieżnym)
- sterowany zdarzeniami: rozgłoszeniowy, sterowania przerwaniami (głównie RTS)

Architektury:

- generyczne – uogólniające doświadczenie z danej dziedziny. Typowe przykłady: przetwarzania rozproszonego, RTS, GUI, systemy przetwarzania danych, systemy sterowania
- referencyjne – stanowią normy dla danej dziedziny np model OSI dla programów sieciowych

31 Diagram komponentów



Rysunek 7: Diagram komponentów UML

32 Klasa a obiekt

Obiektem nazywamy byt istniejący w trakcie wykonywania programu; posiada tożsamość, stan i zachowanie. Klasa jest całkowitą lub częściową definicją obiektów, w trakcie wykonania – zbiorem obiektów o tym samym stanie i zachowaniu. Oprogramowanie w pełni obiektowe jest to oprogramowanie w którym wszystkie byty w trakcie wykonywania są albo obiektami, albo składowymi obiektów lub klas.

33 Cechy OOP

- enkapsulacja – obiekty ukrywają jak najwięcej informacji o swoim stanie i zachowaniu
- dziedziczenie – klasy mogą wykorzystywać inne klasy do dostarczania fragmentów swoich definicji
- polimorfizm – pewne nazwy stosowane w kodzie mogą odpowiadać różnym bytom, decyzji o wyborze dokonuje się automatycznie

34 Identyfikacja klas

Klasy zgodnie z naszą koncepcją analizy definiuje się przez podział odpowiedzialności przydzielonych poszczególnym klasom. Taka idea może być stosowana do określania całej hierachii klas. Inną koncepcją jest wyszukiwanie klas poprzez analizę opisu dziedziny zastosowań i funkcji realizowanych przez system. Przy wyszukiwaniu przydatna jest analiza diagramów komponentów, czynności i stanów. Metody:

1. analiza zmienności i wariantów – wszystkie zmienności i warianty rozwiązań dostarczanych przez funkcję przenosimy do wnętrza klasy np. zamiast

   ```
   rysowanie_figury(figura * fPtr)
   {
   }
   ```

   stosujemy

```
class figura
{
  int rodzaj_figury;
  wierzcholek * wPtr;

  void rysuj_sie()
}
```

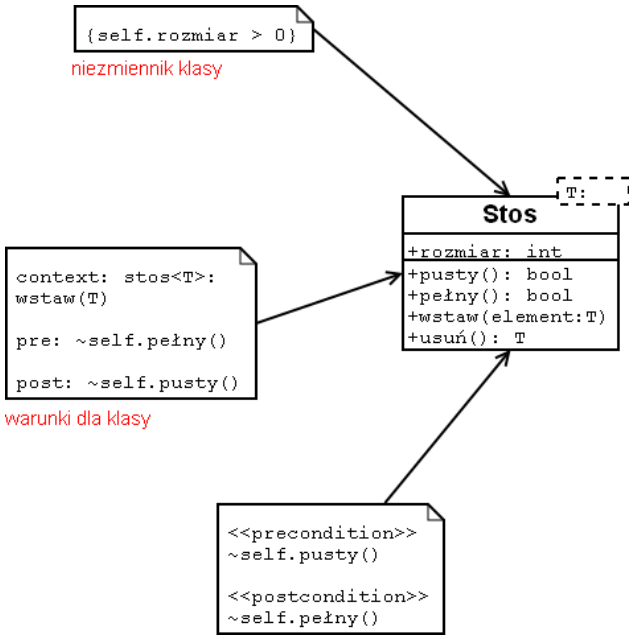
2. podział odpowiedzialności – projektując interfejs uwzględniamy, czy inna klasa nie wykona zadania lepiej i oddelegować wykonanie do innej klasy (np. standard MVC – złożony obiekt składa się z widoku [wyświetla aktualny stan], model [przetwarza zmiany stanu obiektu oraz kontroler [dostosowuje widok do zmiany stanu])

35 Klasa w projektowaniu kontraktowym

Klasa powinna być realizacją dobrze zdefiniowanego, abstrakcyjnego typu danych. Powinna mieć precyzyjny interfejs i być przystosowana do wielokrotnego użycia. Z każdą klasą wiąże się zbiór precyzyjnie określonych asercji – zdań zawsze prawdziwych dla każdego obiektu. Asercje dzielą się na:

- warunki początkowe (*precondition*) – dla każdej metody
- warunki końcowe (*postcondition*) – dla każdej metody
- niezmienniki (*invariants*) – dla każdej klasy

Na schemacie UML asercje umieszczamy w notatkach, odpowiednio oznaczając jak na rysunku 8.



Rysunek 8: Klasa zgodna z zasadami *design by contract*

W projektowaniu kontraktowym awaria wynikła z powodu nie spełnienia warunków początkowych przy wykonaniu operacji nie obciąża twórcy, warunki końcowe określają za co odpowiedzialny jest twórca.

36 Projektowanie kontraktowe a wyjątki i dziedziczenie

Wyjątek w projektowaniu kontraktowym jt. zdarzenie uniemożliwiające wykonanie operacji pomimo spełnienia warunków poczatkowych.

Obiekt klasy pochodnej powinien zostać użyty w miejsce obiektu klasy podstawowej, gdy metody klasy pochodnej mają:

- słabsze lub takie same warunki początkowe
- mocniejsze lub takie same warunki końcowe
- mocniejsze lub takie same niezmienniki

37 Relacje pomiędzy klasami

Związki pomiędzy klasami są oznaczane liniami, które mogą, opcjonalnie, mieć strzałkę wskazującą kierunek relacji. Końce związków mogą także być oznaczone krotnościami: mówią one o tym, ile obiektów danej klasy może brać udział w danej relacji. Na przykład istnieje jeden katalog ale może się nim posługiwać dowolna liczba pracowników biblioteki (co oznaczamy 0...*).

37.1 Zależność (*abstraction*)

Zależność to związek użycia. Zmiany dokonane w specyfikacji jednego elementu klasy (czyli obiektu), np. należącego do klasy Okno, może mieć wpływ na inny element, który używa tego tego pierwszego, ale nie koniecznie odwrotnie. Z zależności należy korzystać korzystać wtedy, kiedy chce się podkreślić, że jeden element używa drugiego. Najczęściej mamy do czynienia z zależnościami pomiędzy klasami, dla pokazania, że jedna klasa używa drugiej jako argumentu w sygnaturze operacji. Zmiany wprowadzone w danej klasie mogą mieć wpływ na operacje innej klasy.

37.2 Asocjacja (*association*)

Asocjacja to związek strukturalny, który wskazuje, że obiekty jednej klasy są połączone z obiektami innej klasy. Asocjacja pomiędzy dwoma klasami oznacza, że przejść od obiektu jednej z klas do obiektu drugiej oraz odwrotnie jest możliwe. Jeśli połączenie dotyczy dokładnie dwu klas, to taką asocjację nazywamy dwuargumentową. Na diagramie asocjacja jest przedstawiona jako linia ciągła.

37.3 Agregacja (*aggregation*)

Agregacja to związek typu „całość-część", w którym jedna z klas reprezentuje „całość", zaś pozostała klasa lub klasy reprezentują „część". Innymi słowy klasa reprezentująca całość składa się z pozostałych klas. Agregacja to szczególny przypadek asocjacji. Prosta postać agregacji ma znaczenie pojęciowe, jak również nie wyznacza zależności między czasem życia „całości" a czasem życia „części".

37.4 Generalizacja (*generalization*)

Generalizacja w ujęciu UML odpowiada ogólnie przyjętej definicji. Jest to związek występujący między bardziej ogólnym elementem (nadklasą, rodzicem) a bardziej szczegółowym elementem (podklasą, dzieckiem) w pełni zgodnym z nadrzędnym i zawierającym ponad to dodatkowe informacje czy własności. Generalizacja posiada predefiniowane komplementarne pary ograniczeń, overlapping i disjoint (względem rozłączności podklas) oraz complete i incomplete (względem wyróżnienia wszystkich podklas). Przydatnym elementem notacji jest dyskryminator pozwalający określić ze względu na jaką własność ma miejsce związek generalizacji np.: drzewo -¿ dąb, brzoza, sosna (dyskryminator = gatunek).

38 Dziedziczenie

39 Zasadność dziedziczenia

Dziedziczenie – klasa pochodna korzysta z definicji klasy podstawowej. Podstawowe rodzaj dziedziczenia to:

- dziedziczenie interfejsu – klasa pochodna dziedziczy wyłącznie funkcje składowe klasy podstawowej
- dziedziczenie interfejsu i implementacji – dziedziczone są dane składowe, sygnatury i implementacja metod

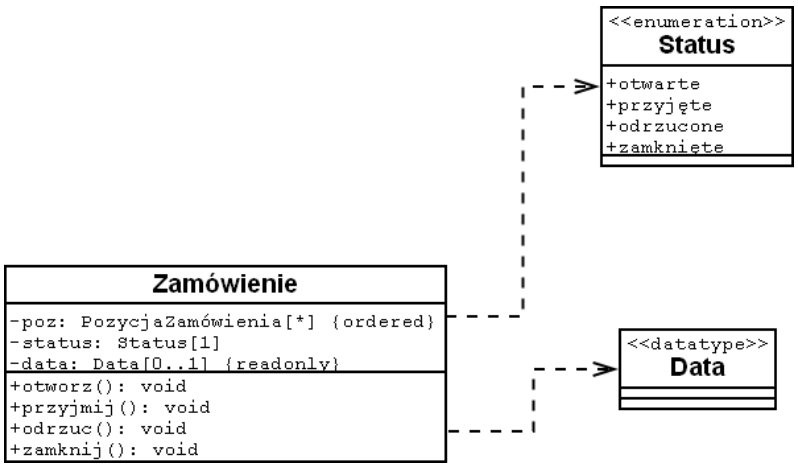
Klasa pochodna powinna wnosić conajmniej jedną z poniższych zmian:

- nowe dane lub funkcje składowe
- redefinicja istniejących funkcji składowych
- zmiana niezmienników klasy

Dziedziczenie jest konieczne gdy:

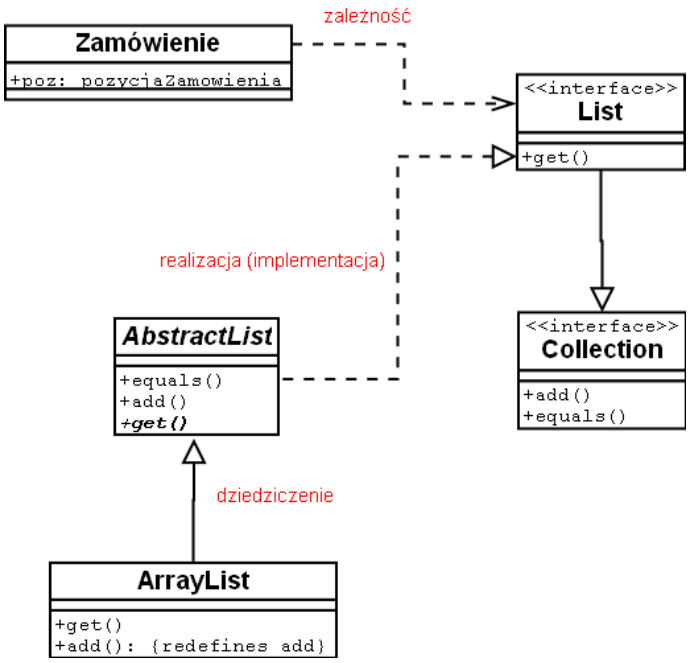
- chcemy używać obiektów klas pochodnych do których typ określać będzie klasa podstawowa
- chcemy wprowadzić nową klasę, która częściowo modyfikuje zachowania lub własności innej klasy, przy czym większość starej klasy pozostaje bez zmian
- nie spodziewamy się, aby obiekt klasy pochodnej miał w trakcie istnienia dokonać zmiany atrybutów i zachowania związanych z klasą podstawową

40 Diagram klasy z pełną specyfikacją



Rysunek 9: Projekt klasy z pełnym opisem

41 Diagram z powiązaniami

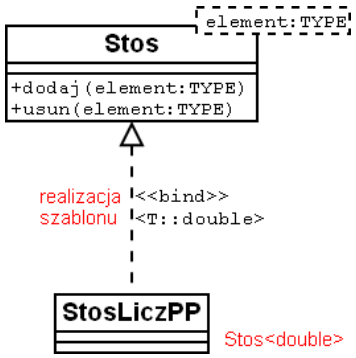


Rysunek 10: Powiązania między klasami

42 Diagram klasy z pełną specyfikacją

(patrz punkt 41.)

43 Klasa parametryzowana



Rysunek 11: Klasa parametryzowana

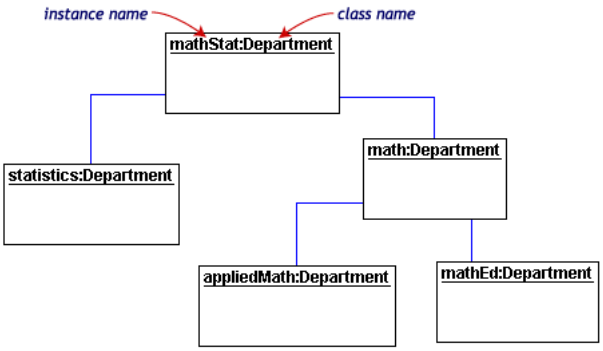
44 Typowe funkcje pomocnicze

Typowe funkcje pomocnicze w niektórych nowoczesnych językach obiektowych (np. C-sharp) mogą zostać zastąpione przez tzw. settery i gettery. Charakterystyczne funkcje pomocnicze dla klas to `element get()` czy `void set(element)`, ustawiające wartości prywatnych zmiennych klasy. Implementacja w języku C-sharp polega na ustawieniu getterów i setterów dla zmiennych prywatnych np.

```
private int _x;

public int x
{
  get { return this._x; }
  set { this._x = value }
}
```

45 Diagram obiektów UML

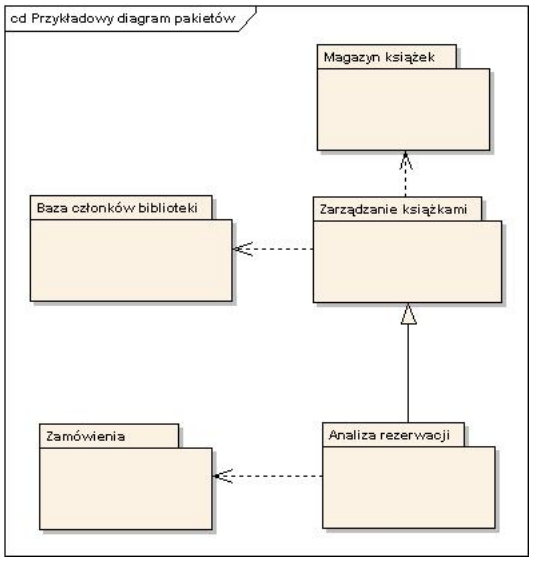


Rysunek 12: Diagram obiektów UML

46 Rola diagramów pakietów UML

47 Diagram pakietów UML

Diagram pakietów służy do tego, by uporządkować strukturę zależności w systemie, który ma bardzo wiele klas, przypadków użycia itp. Przyjmujemy, że pakiet zawiera w sobie wiele elementów, które opisują jakieś w miarę dobrze określone zadanie. Na diagramie umieszczamy pakiety i wskazujemy na zależności między nimi. Dzięki temu dostajemy na jednym diagramie obraz całości, bądź dużego fragmentu, systemu.

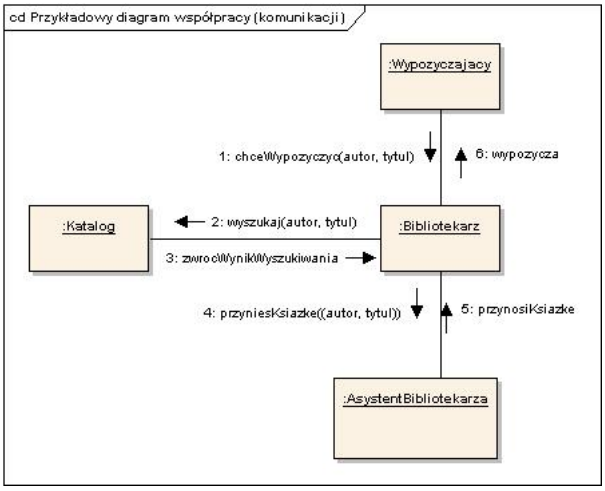


Rysunek 13: Diagram pakietów UML

48 Diagram komunikacji

49 Diagram sekwencji

Diagram współpracy jest jednym z czterech diagramów interakcji. Używamy go po to, żeby zobrazować dynamikę systemu – wzajemne oddziaływanie na siebie obiektów oraz komunikaty, jakie między sobą przesyłają.

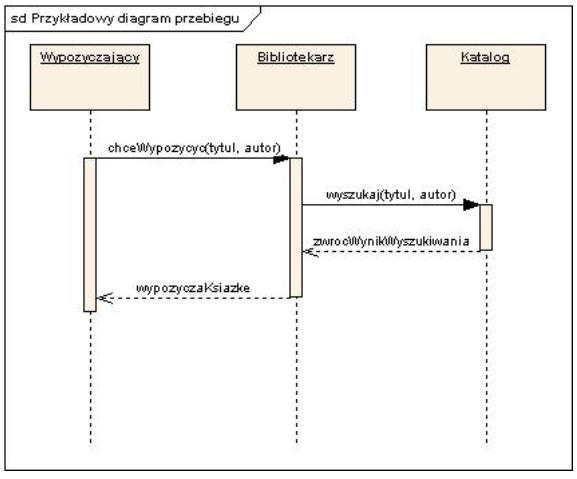


Rysunek 14: Diagram komunikacji UML

Linie, jak zwykle, oznaczają związki między obiektami (prostokąty), strzałki wskazują kierunki przesyłania określonych komunikatów, a numery ich kolejność.

Warto zwrócić uwagę na to, że akurat w tym konkretnym przypadku diagram UML-a został użyty do tego, aby przedstawić pewien element działania biblioteki z bardzo ogólnego punktu widzenia (np. dyrektora biblioteki). Dlatego też związki i komunikaty między obiektami mogą się znacznie różnić od tego, co zrobi później programista implementując system, tak, by był on łatwy w utrzymaniu i rozbudowie.

Analogiczną informację do diagramu komunikacji zawiera drugi z diagramów interakcji, diagram przebiegu. Diagram komunikacji koncentrował się na zobrazowaniu współpracy między obiektami, teraz chcemy pokazać kolejność przesyłania komunikatów i czas istnienia obiektów.



Rysunek 15: Diagram przebiegu UML

50 Poziomy ponownego wykorzystania kodu

- copy and paste – najgorsze rozwiązanie, zwykle stanowi źródło błędów
- techniki programistyczne – dziedziczenie, programowanie generyczne, web services, programowanie aspektowe
- komponenty, biblioteki i toolboxy
- szkielety (frameworks) – oprogramowanie wymagające uzupełnienia w celu uzyskania pełnej funkcjonalności
- całe, gotowe programy – ponowne wykorzystanie możliwe ze zmienioną konfiguracją lub w ramach nowego środowiska

51 Zalety i wady ponownego wykorzystania kodu

Zalety

- redukcja kosztów i czasu
- zwiększenie niezawodności
- zmniejszenie ryzyka związanego z tworzeniem
- popularne komponenty wprowadzają standardy, które ułatwiają tworzenie i stosowanie programów
- w oparciu o wielokrotne użycie można tworzyć generatory programów lub istotnych podsystemów (np. UI, parserów, kodu z UML)

Wady

- nie istnieją sposoby łatwego wyszukiwania istniejącego kodu (na pewno? Google Code Search? CodeFetch?)
- zanim zastosuje się gotowy kod, należy go zrozumieć
- zastosowanie może wymagać istotnego dopasowania systemu
- utrudnione testowanie, gdy istniejący kod przeplata się z nowym
- uzależnienie od kodu nad którym nie panujemy

52 Komponenty

Komponenty są to elementy większe niż pojedyncze klasy, lecz mniejsze niż frameworki. Podstawową cechą jest zdolność do niezależnego montowania bez konieczności kompilacji. Komponent musi być w pełni określony przez interfejs, który realizuje.

Przykłady komponentów: CORBA, EJB, (D)COM(+). Rozwijaną ostatnio alternatywą dla komponentów są *webservices*. Elementy środowisk komponentowych: serwer usług, aplikacja kliencka, middleware. W przypadku webservices występuje jeszcze repozytorium usług.

53 Proces developmentu z użyciem komponentów

- w fazie określania wymagań należy wyszukać komponenty nadające się do zastosowania w systemie
- w fazie projektowania uwzględnia się już wybrane komponenty, a także inne, nie wpływające na wymagania
- w fazie implementacji należy uwzględnić zwiększone zasoby przeznaczone na integrację komponentów
- w każdej z faz należy uwzględnić ryzyko, że komponent okaże się nieodpowiedni

W praktyce dodatkowo opiera się na istnieniu:

- standardów specyfikacji komponentu i wymiany informacji
- platform integracji komponentów (tzw. *middleware*) umożliwiających integrację

54 Zasady tworzenia komponentów wielokrotnego użycia

Tworząc komponenty należy się starać by

- realizować w nich funkcjonalność, która opierać się będzie próbie czasu
- uwzględnić w interfejsie wszelkie aspekty życia komponentu, a jednak nie skomplikować go tak, aby utrudnić jego użycie
- uniezależnić komponent od sprzętu i systemu
- umożliwić stosowanie w różnych sytuacjach przez dodatkową konfigurację
- uczynić komponent maksymalnie samowystarczalnym (poprzez np. statyczne linkowanie bibliotek)

55 Elementy środowisk komponentowych

(patrz pytanie 52.)

56 Wady i zalety środowisk komponentowych

Wady:

- wszystkie z ponownego użycia kodu
- obsługa błędów i sytuacji wyjątkowych – trzeba pamiętać, że w praktyce w konkretnym systemie wykorzystywana jest tylko część funkcjonalności dostarczanej przez komponent
- zaufanie do komponentów, które specyfikowane przez swoje interfejsy są czarnymi skrzynkami, które mogą kryć w swym wnętrzu nieprzyjemne niespodzianki (niezgodność ze specyfikacją, niespełnianie wymagań niefunkcjonalnych, nie mówiąc o zawieraniu szpiegów i wirusów)

Zalety:

- środowisko może samo dostarczać dodatkowe funkcje związane z obsługą komponentów (zapewnienie bezpieczeństwa, współbieżności, obsługi transakcji itp.)

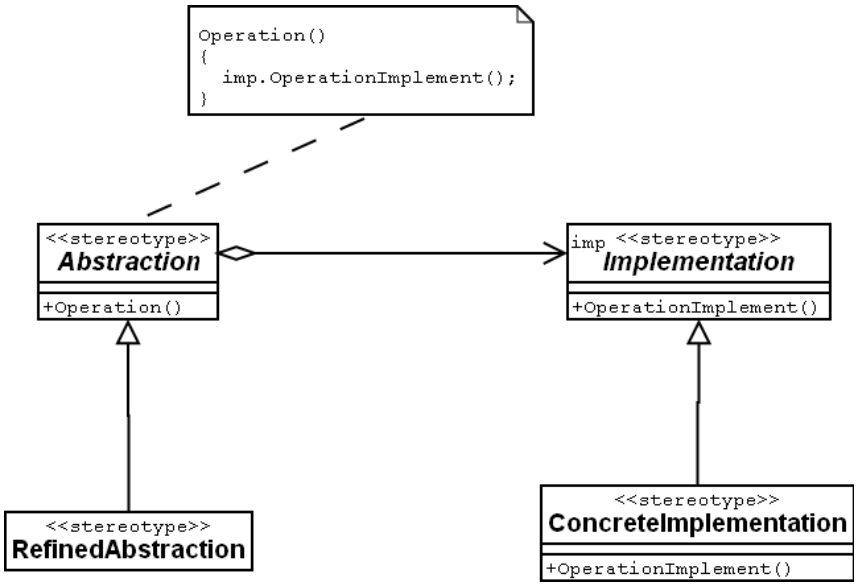
57 Wzorce projektowe

Wzorce projektowe z założenia przedstawiają szkic rozwiązania, który należy uszczegółowić i dostosować do konkretnego kontekstu. Są realizacją koncepcji ponownego użycia, ale w odniesieniu do idei, a nie konkretnego kodu. Rozwiżaują konkretne problemy często pojawiające się w praktyce. Standard prezentacji wzorców:

- nazwa
- przeznaczenie – zwięzła ch-ka motywacji i celu
- inne nazwy
- uzasadnienie stosowania – ogólny opis problemu i jego rozwiązania przy pomocy wzorca
- stosowalność – różne konteksty użycia
- struktura – najczęściej w postaci diagramu UML
- uczestnicy – szczegółowy opis klas i obiektów
- współpraca
- konsekwencje – aspekty zastosowania wzorca
- implementacja
- przykłady
- wzorce pokrewne

58 Przykład wzorca

```
Nazwa: Most
Cel, motywacja: oddzielenie abstrakcji od implementacji
Struktura:
```



Rysunek 16: Wzorzec programowy mostu

59 Zasady projektowania z użyciem wzorców

Wzorce starają się propagować następujące zasady programowania obiektowego:

- programuj pod kątem interfejsów (abstrakcji), a nie implementacji
- przedkładaj składanie obiektów nad dziedziczenie
- wyodrębniaj to co może się zmieniać i hermetyzuj w osobnych klasach
- projektuj klasy tak, aby były otwarte dla rozszerzeń i zamknięte dla modyfikacji (zasada otwarte/zamknięte, open/closed principle)
- staraj się aby klasy miały spójną wewnętrzną strukturę (najlepiej tylko jeden zakres odpowiedzialności), a powiązania z innymi klasami były luźne i elastyczne

60 Refaktoryzacja

Refaktoryzacja jest to poprawa projektu klas i ich struktury bez zmiany funkcjonalności. Podstawowe kieruni refaktoryzacji:

- uproszczenie struktury
- zwiększenie elastyczności
- zmniejszenie dużych klas
- skracanie długich metod

- unikanie klas podobnych do struktur
- unikanie duplikowania
- unikanie konstrukcji *switch* i rozbudowanych *if..else*

Refaktoryzacja jest możliwa dzięki oddzieleniu interfejsu od implementacji. Sprawia, że możliwe staje się ewolucyjne tworzenie oprogramowania, które pozostaje czytelne i proste.

61 Programowanie aspektowe

Programowanie aspektowe zakłada, że w programach pojawiają się elementy dwóch typów:

- elementy dające się hermetyzować w jednostki o ściśle określonej funkcjonalności, dobrze oddzielone od innych
- elementy typu „cross-cut", które przecinają jednostki hermetyzacji i określają pewien aspekt grupy jednostek np. organizacja dostępu, komunikacja, synchronizacja, obsługa błędów itp.

Sposobem osiągnięcia i realizacji programowania aspektowego jest określenie tzw. punktów łączenia (*join points*), w których modyfikacja jest możliwa. Innym sposobem jest możliwość kontroli danego aspektu dla całej grupy jednostek.

- wyrażenie działań (*advices*) realizowanych w momencie osiągnięcia punktu połączenia w danej jednostce
- wykryce punktów łączenia (*pointcut*) i realizacja przewidzianych działań dokonywane są często w trakcie wykonania kodu

Programowanie aspektowe jest związane z tzw. **mechanizmem refleksji** za pomocą którego program może modyfikować swoje zachowania w trakcie wykonania. Programowanie aspektowe wprowadza szereg nie rozwiązanych problemów z wydajnością i bezpieczeństwem kodu.

62 Elementy typowego środowiska RAD

Przykład typowego środowiska RAD – interfejs bazy danych, narzędzia do generowania raportów. Środowisko takie zwykle zawiera:

- narzędzia interfejsu z bazą
- generator interfejsu użytkownika
- narzędzia generowania raportów
- powiązania z arkuszami kalkulacyjnymi

63 RAD jako technologia

RAD jest to technika szybkiego konstruowania programów, w których istotną rolę odgrywa UI, natomiast mniejszą przetwarzanie danych. Programy mają zbliżoną strukturę, wiele czynności daje się zautomatyzować. Typowe jest korzystanie z narzędzi CASE oraz posługiwanie się programowaniem graficznym (*visual programming*). RAD zakłada tzw. *time boxing* – tworzenie oprogramowania w przedziałach czasowych o ściśle okreslonej długości. Oprogramowanie tworzone jest przez mały zespół, który wytwarza oprogramowanie jako serię prototypów. Tworzenie opiera się głównie na predefiniowanych elementach, API, bibliotekach, komponentach etc. **Wady**:

- obniżenie standardów jakości
- zależność od komponentów – kłopoty z ewolucją oprogramowania, kosztami itp.
- stosowanie komponentów prowadzi do niekompatybilności wymagań
- programy stają się do siebie zbyt podobne

Techniki RAD mogą przydać się przy określaniu wymagań oraz projektowaniu w większych projektach.

64 Miary niezawodności systemów informatycznych

- prawdopodobieństwo wystąpienia awarii
- średni czas pomiędzy awariami MTBF
- dostępność – procent czasu w jakim system pozostaje do dyspozycji użytkownika

65 Sposoby gwarantowania niezawodności oprogramowania

Unikanie błędów poprzez

- stosowanie projektowania kontraktowego i włączenie go do kodu
- obsługę wyjątków
- dbanie o czytelność i prostotę kodu
- unikatnie konstrukcji często prowadzących do błędów
- stosowanie ukrywania informacji

Podstawowe mechanizmy uzyskiwania odporności na błędy:

- redundancja – nadmiarowość elementów
- dywersyfikacja – zróżnicowanie elementów
- oba łącznie

66 Sposoby uzyskiwania odporności na błędy

(jak w punkcie 65.)

Systemy odporne na błędy charakteryzują się redundancją i dywersyfikacją włącznie, co oznacza, że system często pracuje jako kilka równoległych systemów, dostarczanych przez różne firmy, działające na różnej zasadzie, lecz zwracające te same wyniki. Wyniki z wszystkich systemów są porównywane przy pomocy jakiegoś elementu logicznego i akceptowane jeśli wszystkie są takie same. Wadą takiego rozwiązania jest fakt, że wymagana jest wysoka niezawodność elementu logicznego, gdyż jego błąd niweczy nasze działania zwiększania odporności. Przykładem sprzętowego rozwiązania systemu odpornego na błędy jest macierz RAID (Redundant Array of Independent Disks).

67 Programowanie defensywne

Wszystkie zasady i praktyki w dziedzinie niezawodności oprogramowania można podzielić na cztery grupy:

1. unikanie defektów,
2. wykrywanie defektów,
3. poprawianie defektów oraz
4. tolerowanie defektów.

Unikanie defektów to stosowanie zasad i technik zmniejszające liczbę błędów. Wykrywanie defektów polega na takich funkcjach wewnętrznych, które wykrywają obecność błędów. Poprawianie defektów to tworzenie w oprogramowaniu środków usuwających szkody wynikające z błędów. Tolerowanie defektów to zdolność systemu do działania w obecności błędów.

Z tych czterech podstawowych technik największą wartość ma unikanie defektów. Projektując program nie powinniśmy jednak nigdy zakładać, że będzie on wolny od błędów. Jeżeli zakładamy realistycznie, że program będzie zawsze zawierał (niewykryte dotychczas) błędy, to wykrywanie defektów nabiera wagi podstawowej. Dopiero wówczas, gdy potrafimy wykryć błąd, będziemy mogli go poprawić lub tolerować.

Metody wykrywania defektów możemy podzielić na bierne i czynne. Bierne wykrywanie defektów polega na wykryciu symptomów defektów podczas wykonywania funkcji programu. Czynne wykrywanie defektów polega na samokontroli przeprowadzanej przez program w celu wykrycia defektów.

Jedną z metod czynnego wykrywania defektów jest stosowanie technik programowania defensywnego. Podstawową przesłanką programowania defensywnego jest przekonanie, że najgorszą rzeczą, którą podprogram może zrobić, jest przyjęcie niepoprawnych danych i przekazanie niepoprawnego, lecz wiarygodnego wyniku.

Tworząc środki służące wykrywaniu błędów należy oprzeć się o następujące zasady:

1. Wzajemna podejrzliwość. Zakłada się, że każdy podprogram jest nieufny w stosunku do innego podprogramu i podejrzewa go o błędy. Kiedy podprogram otrzymuje dane od innego podprogramu lub z zewnątrz systemu, powinien zakładać, że dane mogą być niepoprawne i usiłować znaleźć omyłki w danych.
2. Natychmiastowe wykrycie. Błędy należy wykrywać możliwie jak najszybciej po ich wystąpieniu. Zmniejszymy w ten sposób możliwe straty.
3. Redundancja. Wszelkie metody wykrywania błędów opierają się na istnieniu jawnej lub ukrytej redundancji.

Można się przy tym kierować następującymi praktycznymi wskazówkami:

- W miarę możliwości należy sprawdzać atrybuty każdej danej. Jeżeli napis powinien mieć określoną długość to należy sprawdzić, czy ją ma. Jeżeli liczba nie powinna być ujemna, należy zbadać jej znak.
- Jeżeli dana ma pole służące do samoidentyfikacji to należy użyć go do sprawdzeń.
- Należy sprawdzać, czy dane mieszczą się we właściwych im granicach. Na przykład jeżeli dana jest kodem jednego z 5. produktów, to nigdy nie należy zakładać, że jeżeli wartość danej nie jest numerem produktu 1 do 4, to z pewnością jest kodem produktu 5.
- Jeżeli w danych istnieją jawne redundancje to należy je sprawdzać.

- Jeżeli w danych nie ma redundancji to należy dążyć do ich wprowadzenia.
- Należy sprawdzać poprawność danych zewnętrznych (dane programu) i wewnętrznych (zmienne robocze programu).

Powyższe rozważania teoretyczne w praktyce mają dwie pozorne wady: ich stosowanie wymaga zakodowania dodatkowych instrukcji (podprogramów) sprawdzających, co wydłuża czas kodowania programu, oraz powoduje powstanie większego programu wynikowego.

Pierwszy z tych argumentów można z łatwością obalić. Dodatkowy czas poświęcony na pisanie kodu wykrywającego defekty zostanie odzyskany z nawiązką podczas testowania programu. Dzięki zastosowanym technikom o wiele szybciej i dokładniej będzie można zlokalizować błędy. Ponadto program będzie lepiej przetestowany, dzięki czemu zmniejsza się niebezpieczeństwo powstania awarii podczas pracy programu u jego przyszłego użytkownika. Oprócz trudnego do wyceniania efektu psychologicznego (zrażanie użytkownika) ryzykujemy często spore koszty związane z usuwaniem skutków awarii u klienta — zwłaszcza jeżeli mamy więcej użytkowników programu rozsianych na dużym terytorium (koszty dojazdu, telefonicznych rozmów wyjaśniających i tym podobne).

Inaczej ma się sprawa ze zbyt wielkim kodem programu. Rzeczywiście dosyć często programiści wyposażani są w szybkie i rozbudowane komputery dysponujące dużymi i szybkimi dyskami oraz dużą pamięcią operacyjną. Mogą oni sobie pozwolić na pisanie programów wymagających dużej pamięci operacyjnej. Stawianie takich samych wymagań docelowym użytkownikom programu powoduje zwiększenie kosztów (na sprzęt) użytkownika i w efekcie może spowodować zmniejszenie liczby użytkowników. Powstaje zatem dylemat: pisać program bezpieczniejszy czy mniejszy. Języki takie jak Delphi czy C (C++) pozwalają jednak rozwiązać i ten dylemat.

W wielu miejscach w programie zachodzą pewne, zawsze spełnione relacje między wartościami zmiennych. Zawsze spełnione oznacza: gdy program osiągnie dany etap, warunek dla niego określony jest spełniony bez względu na to, co działo się przedtem. Relacje te nazywamy asercjami lub niezmiennikami.

68 Walidacja i weryfikacja

Walidacja jest procesem mającym na celu określenie, czy zaprojektowany i zaimplementowany system spełnia wszystkie wymagania wykrystalizowane w czasie współpracy z klientem w fazie określania wymagań. Weryfikacja ma na celu określenie, czy system działa prawidłowo. Obie powinno się przeprowadzać we wszystkich fazach wytwarzania oprogramowania oraz realizowane są na dwa uzupełniające się sposoby:

- inspekcja wymagań, projektu, wreszcie kodu
- testowanie programu

69 Inspekcja kodu

Inspekcja kodu polega na przeglądzie kodu pod kątem poprawności i spełnienia wszystkich kryteriów. Proces prowadzi do wykrycia większej ilości błędów niż standardowe testy. Prawidłowo przeprowadzona inspekcja kodu przebiega zgodnie z ustalonymi standardami. Wyróżnia się szereg ról, które powinny być realizowane przez osoby dokonujące inspekcji:

- autor kodu – odpowiedzialny za poprawę błędów
- inspektor – wykrywa błędy
- lektor – prezentuje kod

- sekretarz – zapisuje wyniki
- moderator – organizuje proces inspekcji

Proces inspekcji oprócz zebrań obejmuje fazę przygotowania do zebrań oraz fazę realizowania konsekwencji zebrań. Podstawą procesu inspekcji jest ścisła specyfikacja kodu oraz gotowy kod źródłowy, z którymi zapoznają sie osoby dokonujące inspekcji w fazie przygotowania. Podczas zebrań, trwających ok. 2 godziny, przegląda się ok. 200 linii kodu (wg zaleceń). Przegląd może być przeprowadzany w oparciu o listy standardowych błędów popełnianych w programach. Starannie przeprowadzane inspekcje mogą wyeliminować konieczność (lub ekonomiczną opłacalność) testów jednostkowych (komponentów).

70 Rodzaje testów

70.1 Podział testów

Ze względu na cel:

- w ramach walidacji
- w ramach weryfikacji

Ze względu na zakres:

- jednostkowe (poszczególnych komponentów)
- systemu

Ze względu na sposób realizacji:

- losowe
- specjalnie zaprojektowanych przypadków

Ze względu na podejście:

- funkcjonalne – oparte na podziale danych wejściowych na klasy
- strukturalne – zakładające testowanie wszystkich fragmentów kodu

W przypadku testów losowych generujemy dane z zakresu określonego z rozkładem prawdopodobieństwa, co pozwala na oszacowanie niezawodności, lecz w miarę przeprowadzania testów prowadzi do nasycenia – nowe testy wykrywają coraz mniej błędów.

Poprawienie skuteczności testów polega na planowaniu ich zakresu.

71 Składniki wydania

Typowe wydanie (*release*) programu zwykle składa się z:

- program
- pliki konfiguracyjne
- pliki danych do przykładów
- dokumentację (instrukcję instalacji, opis systemu, podręczniki użytkownika i administratora, opis wydania, itp.)

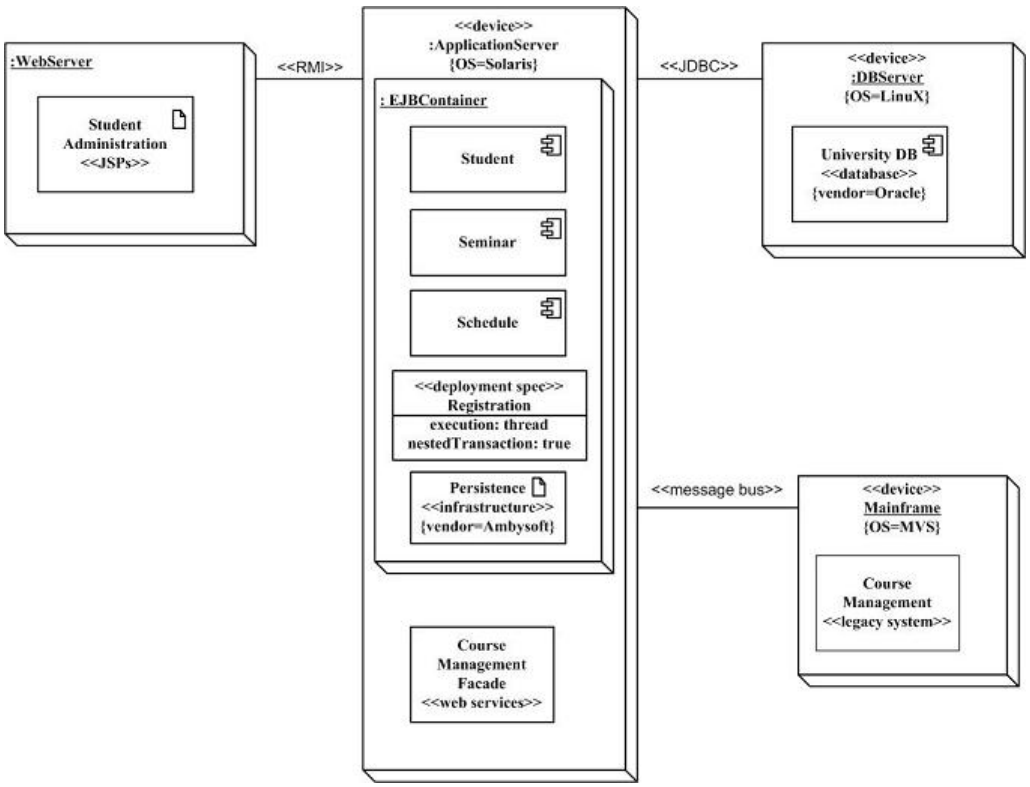
72 Elementy procesu wdrożenia

Wdrożenie obejmuje:

- dostarczenie oprogramowania i (opcjonalnie) sprzętu
- skonfigurowanie sprzętu i oprogramowania
- uruchomienie systemu

Wdrożenie może być dokonywane przez klienta (zazwyczaj przy użyciu odpowiednich narzędzi), niezależną od producenta firmę, wreszcie przez przedstawicieli producenta. Dokonanie wdrożenia połączone jest często z przeprowadzeniem szkolenia pracowników użytkujących i obsługujących system. Wdrożenie może być złożonym procesem jeśli ostateczna konfiguracja komponentów jest jednostkowa dla danego przypadku i dokonywana bezpośrednio u klienta. Firmy dokonujące wdrożenia powinny realizować zarządzanie wersjami i konfiguracją (z użyciem odpowiedniej bazy danych), aby umożliwić dalszą konserwację systemu.

73 Diagram wdrożenia UML



Rysunek 17: Diagram wdrożenia UML

74 Konserwacja oprogramowania

Typowe czynności związane z konserwacją obejmują:

- przyjmowanie zgłoszeń o błędach i propozycji zmian w funkcjonowaniu systemu
- oszacowanie koniecznych zmian w kodzie (impact analysis)
- zaplanowanie harmonogramu wprowadzania zmian
- wprowadzenie zmian
- przygotowanie poprawek lub kolejnego wydania

Typowe powody modyfikacji programów to:

- zmiany wymagań (ok. 65 proc.)
- dostosowanie do zmian technologicznych w sprzęcie i jego oprogramowaniu systemowym (ok. 18 proc.)
- naprawa błędów (ok. 17 proc.)

75 Cechy charakterystyczne ewolucji systemów

W toku funkcjonowania wielu dużych systemów informatycznych zaobserwowano, że podlegają one szeregowi prawidłowości:

- programy stale ewoluują, aby pozostać w pełni użyteczne; dostosowują się do zmian technologicznych i rozszerzają swoją funkcjonalność
- w miarę ewolucji programy stają się coraz większe i coraz bardziej złożone
- jakość programów ma tendencję do degradacji w miarę ich wzrostu...
- chyba że istnieje wdrożony rozbudowany system informacji zwrotnej o błędach (obejmuje to także zgłaszanie proponowanych zmian w systemie, requests for change) i mechanizmy korekty systemu

76 Zarządzanie projektem informatycznym a inne projekty

Zarządzanie projektem informatycznym ma wiele cech wspólnym z zarządzaniem innym dowolnym przedsięwzięciem:

- systemy są często nowatorskie i jednorazowe, co utrudnia identyfikację wymagań
- przy tworzeniu łatwo popełnia się błędy
- oprogramowanie nie daje się jednorazowo „oglądnąć", przez co trudno śledzi się tok rozwoju
- brak sprawdzonych standardów wytwarzania

Ważne jest szacowanie ryzyka i opracowanie sposobów radzenia sobie w różnych wariantach rozwoju sytuacji.

77 Podstawowe czynności związane z zarządzaniem projektem

- planowanie (specyfikacja celu i efektu końcowego oraz efektów pośrednich, ustalenie składu zespołu realizującego, określanie harmonogramu, szacowanie kosztu, analiza ryzyka, określenie wymagań sprzętowych, itp.)
- nadzorowanie realizacji (zarządzanie ludźmi, jakością oraz ryzykiem)
- dokumentacja + prezentacja osiągniętych wyników

78 Analiza ryzyka

Analiza ryzyka obejmuje wykrywanie potencjalnych zagrożeń dla planowanej realizacji przedsięwzięcia, szacowanie prawdopodobieństwa ich wystąpienia, określanie sposobów radzenia sobie z nimi i ostateczne ustalanie oczekiwanych skutków dla realizacji całości. Przykłady zagrożeń są na tyle liczne, że trudno je wymienić i obejmują m.in.:

- ryzyka dotyczące personelu
- ryzyka zmian wymagań
- ryzyka zmian technologicznych
- ryzyka zmian organizacyjnych
- ryzyka konkurencji
- ryzyka związane z narzędziami

79 Szacowanie kosztu

Ze względu na złożoność realizacji projektów informatycznych (i konieczność uwzględnienia wielu nieplanowanych zdarzeń) szacowanie kosztu wytwarzania oprogramowania jest trudne (podobnie jak ustalanie harmonogramu). Koszt realizacji projektu informatycznego składa się głównie z:

- kosztów osobowych (wynagrodzenie wykonawców i personelu wspomagającego)
- kosztów sprzętu i oprogramowania usługowego
- kosztów realizacji: wyjazdy, szkolenia, utrzymanie pomieszczeń, materiały biurowe, itp.

Najtrudniejszym elementem szacowania kosztu realizacji przedsięwzięcia informatycznego jest określenie wydajności pracy (czyli np. ile zajmie realizacja określonego zadania przez określoną liczbę osób w określonych warunkach). W celu ułatwienia szacowania kosztu wprowadza się rozmaite metryki wydajności:

- metryki oparte na rozmiarze kodu (np. liczba linijek kodu produkowanego miesięcznie przez pojedynczego programistę)
- metryki oparte na uzyskiwanej funkcjonalności (np. interakcjach z użytkownikiem, interakcjach z urządzeniami we/wy)

Metryki są bardzo szacunkowe i przybliżone, nie uwzględniają rozmaitości i złożoności oprogramowania.

80 Wspomaganie szacowania kosztu

Szacowanie kosztu można wspomagać następującymi technikami:

- modelowaniem algorytmicznym
- ekspertyzami specjalistów
- porównaniem z uprzednio realizowanymi projektami
- ograniczeniami np. warunki zamówienia, stan firmy

81 Wady i zalety algorytmicznego szacowania kosztu

Model COCOMO (Cost Construction Model) stosuje proste wzory matematyczne z szeregiem parametrów:

$$Koszt = a * miara^b$$

Model COCOMO jest modelem empirycznym powstałym z uogólnienia dotychczasowych doświadczeń z wytwarzaniem oprogramowania. W modelu COCOMO II wyróżnia się cztery podmodele dla różnych faz projektu: kompozycyjny (application composition), fazy wstępnej (early design), ponownego użycia (reuse) i post-architektoniczny (post-architecture). Dobór parametrów w modelu COCOMO jest bardzo złożony — każdy ostateczny parametr jest sumą lub iloczynem wielu innych parametrów cząstkowych. Stosowanie modelu COCOMO, pomimo jego szczegółowości, pozostawia jednak wiele miejsca na subiektywne oceny i jest podatne na błędy i niedokładności. Sposobem redukcji arbitralności w modelu COCOMO może być jego lokalna kalibracja na podstawie uprzednich projektów. Choć stosowanie modelu COCOMO powinno odbywać się z dużą ostrożnością, jego zaletą jest zgromadzenie w jednym algorytmie rozmaitych czynników wpływających na nakłady, koszt i harmonogram projektów informatycznych.

82 Zarządzanie jakością

Aby uzyskać oprogramowanie wysokiej jakości przydatne (konieczne?) jest:

- wprowadzenie w przedsiębiorstwie procedur pomagających uzyskać wysoką jakość
- planowanie uzyskania wysokiej jakości w konkretnym projekcie
- nadzorowanie realizacji projektu zgodnie z wymaganiami wysokiej jakości

Wprowadzenie procedur gwarantujących jakość produktów jest związane z zarządzaniem procesem rozwoju oprogramowania w firmie. Firmy często wprowadzają własne standardy, np. pisania kodu (coding standards). Dla oceny jakości oprogramowania i jakości procesu jego wytwarzania wprowadza się rozmaite miary:

- miary dynamiczne, związane z wykonaniem programów i odpowiadające jakości oprogramowania
- miary statyczne dotyczące samego kodu, takie jak:
  - rozmiar kodu
  - ilość wywołań pojedynczej procedury i ilość wywołań w pojedynczej procedurze (fan-in, fan-out)
  - długość nazw

- głębokość instrukcji warunkowych
- głębokość hierarchii dziedziczenia
- liczba metod w klasie
- liczba przesłoniętych metod i operacji

Elementem uzyskiwania wysokiej jakości oprogramowania jest prawidłowa dokumentacja procesu wytwarzania i samego oprogramowania. Dokumentację można i powinno się (wymagają tego np. normy ISO9000) prowadzić na wszystkich etapach wytwarzania. Dokumentacja jest najczęściej tworzona zgodnie z pewnymi standardami (także udokumentowanymi).

83 Całościowa dokumentacja projektu

84 Finalna wersja dokumentacji projektu

W fazie realizacji procesu wytwarzania oprogramowania dokumentacja może obejmować:

- plany, harmonogramy, szacunki
- wszelkiego typu projekty i modele systemu oraz jego elementów (różnorodne diagramy, schematy, algorytmy) związane z wszystkimi fazami procesu wytwarzania oprogramowania
- raporty
- kod źródłowy
- korespondencja, wymiana informacji na temat projektu między członkami zespołu realizującego

W skład dokumentacji końcowej (dostarczanej odbiorcom) najczęściej wchodzą:

- ogólny funkcjonalny opis systemu
- przewodnik użytkownika (user's guide)
- pełny opis systemu (reference manual)
- przewodnik instalacji i podręcznik administratora
- dokumentację wersji (wydania, release notes)

Zaletą dobrej dokumentacji jest posiadanie szczegółowego spisu treści, słownika pojęć i bogatego indeksu. Coraz częściej dokumentacja dostarczana jest wyłącznie w formie elektronicznej wraz z systemem. Alternatywnie dokumentacja dostępna jest przez sieć (vide: MSDN Library)

85 Model CMM

Model CMM został wprowadzony przez amerykańską organizację SEI (Software Engineering Institute) w celu zwrócenia uwagi na znaczenia jakości procesu wytwarzania oprogramowania dla jakości samego wytwarzanego produktu. W ramach modelu CMM procesy zaliczane są, na podstawie rozmaitych kryteriów, do jednego z pięciu poziomów:

- wstępny (initial)
- powtarzalny (repeatable)
- zdefiniowany (defined)
- ilościowo zarządzany (quantitatively managed)
- optymalizujący (optimizing)

86 RUP

RUP jest metodologią tworzenia oprogramowania powiązaną ze spiralnym modelem cyklu życia oprogramowania oraz z wcześniejszymi metodami, które legły u podstaw powstania języka UML (OMT, OOSE). RUP definiuje szkielet postępowania, który należy dostosować do uwarunkowań konkretnego projektu programistycznego. RUP powstał na bazie analizy najczęstszych przyczyn niepowodzeń istniejących procesów wytwarzania oprogramowania. Oparty jest o pewne podstawowe zasady oraz fazy wytwarzania oprogramowania. Zasady RUP:

- iteracyjne i przyrostowe tworzenie oprogramowania; sterowane ryzykiem i priorytetami, ułatwiające integrację całości kodu i dostosowanie do zmieniających się wymagań
- zarządzanie wymaganiami; we współpracy z klientem i w oparciu o przypadki użycia
- stosowanie architektury opartej na komponentach
- graficzne modelowanie oprogramowania; różne perspektywy spojrzenia na system, użycie UML
- kontrola i weryfikacja jakości oprogramowania przez cały czas procesu wytwarzania
- zarządzanie zmianami w oprogramowaniu

Fazy projektu RUP:

- faza początkowa (inception) – wstępne określenie wymagań, ryzyka, kosztu, harmonogramu, a także architektury systemu
- faza opracowania (elaboration) – ustalenie wymagań (większości przypadków użycia), architektury systemu oraz planu całego procesu wytwarzania systemu
- faza konstrukcji (construction) – tworzenie systemu (kolejnych komponentów), w trakcie nastepuje oddanie pierwszej (i być może dalszych) wersji użytkownikowi
- faza przekazania (transition) – system jest przekazywany użytkownikowi, wdrażany, szkoleni są pracownicy obsługi systemu, nastepuje walidacja i końcowe sprawdzenie jakości

RUP wyróżnia także „dyscypliny”, grupy zadań wykonywanych przez pracowników:

- modelowanie biznesowe
- wymagania
- analiza i projektowanie
- implementacja
- testowanie
- wdrożenie
- zarządzanie konfiguracją i zmianami
- zarządzanie projektem (zarządzanie ryzykiem, planowanie iteracji, monitorowanie postępów)
- organizacja środowiska (m.in. narzędzi)

87 Extreme programming

Podstawowe zasady XP:

- oprogramowanie jest rozwijane w krótkich cyklach i w ciągłej interakcji z klientem, po każdym cyklu może nastąpić zmiana założeń co do dalszej pracy, co więcej może nastąpić rewizja już napisanego kodu (refactoring)
- każdy przyrost dostarcza konkretną funkcjonalność określaną na podstawie scenariuszy („opowieści użytkownika", user stories)
- już pierwsze wydanie zawiera system o pewnej całościowej strukturze realizujący istotne dla klienta funkcje
- kolejne wydania (releases) kodu następują co kilka miesięcy (maximum), iteracje mają po kilka tygodni, obowiązuje planowanie zadań kilka dni do przodu, przy założeniu, że kolejność działań jest określana przez priorytety ważności
- przed przystąpieniem do kodowania opracowywane są szczegółowe testy mające za zadanie sprawdzenie wszelkich aspektów poprawności wprowadzanych zmian
- każda zmiana jest od razu integrowana z całością kodu oraz testowana – nowe oraz opracowane wcześniej testy (zautomatyzowane) są powtarzane codziennie (lub nawet kilka razy dziennie), żeby sprawdzić czy nie zostały wprowadzone błędy
- w ciągu całego procesu gromadzona jest informacja zwrotna służąca usprawnieniu procesu i ostatecznego kodu
- ważna jest specyficzna organizacja miejsca pracy
- kodowanie realizowane jest zgodnie z przyjętymi przez cały zespół regułami (coding standards)
- programowanie odbywa się parami (zgodnie ze szczegółowo ustalonymi zasadami), częste są także dyskusje pomiędzy członkami całego zespołu tworzącego kod
- dokumentacja oprogramowania składa się z: komentarzy w kodzie, opisu testów.

Jedną z podstawowych różnic pomiędzy programowaniem ekstremalnym i podejściem klasycznym jest stosunek do specyfikacji i dokumentacji. W XP specyfikacja jest rozwijana na bieżąco, w trakcie iteracyjnego tworzenia kodu i we współpracy z klientem (a nie jest tworzona przed implementacją i opisywana w rozległej dokumentacji). Podobnie zapewnienie jakości odbywa się poprzez testowanie dokonywane na bieżąco, a nie poprzez sprawdzenie zgodności ze specyfikacją. Klasyczne podejścia mogą stosować XP do tworzenia prototypów (do wyrzucenia), np. w fazie odkrywania i walidacji wymagań lub projektowania GUI.

Do problemów, na które natyka się programowanie ekstremalne należą:

- niedostosowanie do wymagań niektórych klientów (wymagających kontraktów, rozbudowanej specyfikacji, bogatej dokumentacji itp.)
- niedostosowanie do struktur organizacyjnych tradycyjnych (dużych) firm software'owych (mających np. biura rozproszone w różnych częściach świata)
- niedostosowanie do mentalności niektórych programistów
- trudność utrzymania prostej struktury kodu, w miarę jego wzrostu dokonywanego metodami ewolucyjnymi
- trudność zapewnienia wymagań bezpieczeństwa i niezawodności dla niektórych rodzajów systemów

88 Agile Manifesto

Individuals and interactions over processes and tools. Working software over comprehensive documentation. Customer collaboration over contract negotiation. Responding to change over following a plan.

czyli:

- Jednostki i interakcje ponad procesy i narzędzia
- Działające oprogramowanie ponad wyczerpującą dokumentację
- Współpraca z klientem ponad negocjacje kontraktu
- Reagowanie na zmiany ponad realizowanie planu