Fazy analizy (modelowania) oraz projektowania

Analiza → bez brania pod uwagę szczegółów implementacyjnych
Projektowanie → ze szczegółami implementacyjnymi.

FAZA ANALIZY:

Celem fazy analizy jest ustalenie wszystkich tych czynników, które mogą wpłynąć na przebieg procesu projektowego i implementacyjnego

Wynikiem jest **logiczny model systemu**, opisujący sposób realizacji przez system postawionych wymagań, lecz abstrahujących od szczegółów implementacyjnych

Model oprogramowania powinien być jego uproszonym opisem, opisującym wszystkie istotne cechy oprogramowania na wysokim poziomie abstrakcji.:

- jest zorganizowany hierarchicznie, wg poziomów abstrakcji
- unika terminologii implementacyjnej
- pozwala na wnioskowanie o związkach przyczynowo-skutkowych

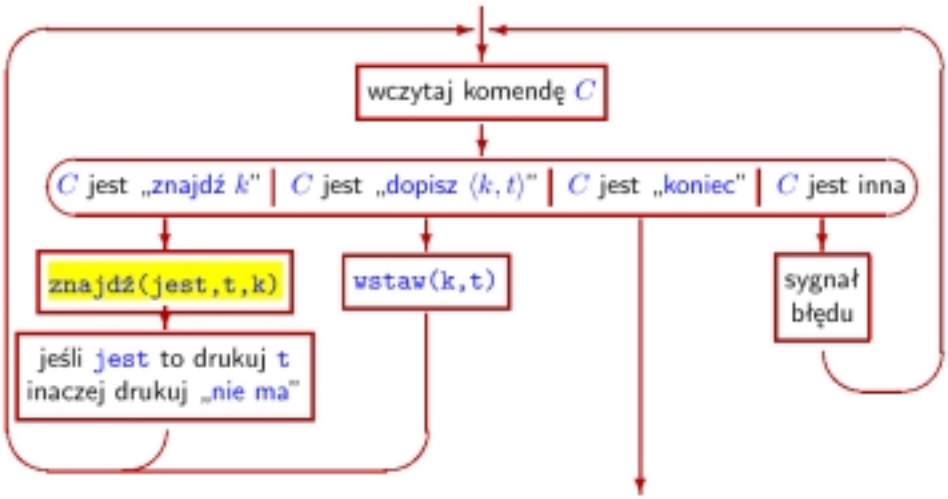
Dobrej jakości model powinien być:

- **kompletny** – wyrażone są w nim wszystkie cechy opisywanego zagadnienia
- **prawidłowy** – wszystkie występujące pojęcia zostały właściwie użyte,
- **minimalny**, jeżeli każdy z aspektów wymagań występuje na schemacie tylko jeden raz,
- **wyrazisty** jeżeli wymagania analizowanego obszaru są reprezentowane na schemacie w naturalny sposób, nie wymagają dodatkowych wyjaśnień,
- **czytelny** jeżeli graficzna reprezentacja zawiera minimum punktów percepcji (przecięć, załamań linii, itp.).,
- podatny na modyfikacje.

Rodzaje notacji dla potrzeb modelowania:

- język **naturalny**,
- zapis **ustrukturalizowany** tekstowo-numeryczny,
- notacje **graficzne**.

Szczególne znaczenie maja notacje graficzne. Ich zalety potwierdzają badania psychologiczne. Na przykład diagram przebiegu sterowania:



Takie opisy i schematy modelują część programową, czyli sterowanie programu. Można je stosować z różnymi poziomami szczegółowości.

STRATEGIE BUDOWY MODELU

Strategia top-down
Od ogółu do szczegółu - najpierw definiuje się ogólne pojęcia, a następnie rozwija się je poprzez dodawanie szczegółów stosując elementy podstawowe (prymitywy).

Strategia bottom-up
Od szczegółu do ogółu - najpierw definiuje się pojęcia elementarne, a następnie buduje się z nich struktury w celu stworzenia pojęć ogólnych.

Strategia inside-out
Rozprzestrzenianie - najpierw definiuje się pojęcia, które wydają się być najważniejsze, a następnie rozwija się je poprzez dobudowywanie kolejnych pojęć, które stanowią ich uzupełnienie.

W praktyce strategie analizy (i projektowania) są zwykle oparte na rozprzestrzenianiu, z zastosowaniem podejścia *top-down* lub *bottom-up* w ramach tworzonych fragmentów systemu.

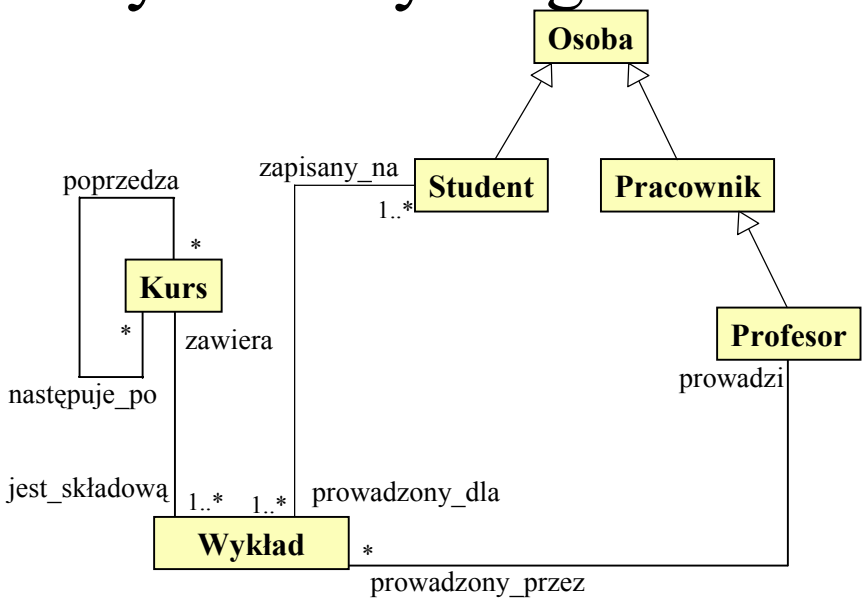
UML (Unified Modeling Language) jest metodyką projektowania, tzn. zestawem pojęć, oznaczeń, diagramów oraz zaleceń praktycznych. Notacja UML, która opiera się o podstawowe pojęcia obiektowości może być wykorzystana w dowolnej metodyce.

Diagramy definiowane w UML:

- Diagramy przypadków użycia
- Diagramy klas, w tym diagramy pakietów
- Diagramy zachowania się:
  - Diagramy stanów
  - Diagramy aktywności
  - Diagramy sekwencji
  - Diagramy współpracy
- Diagramy implementacyjne:
  - Diagramy komponentów
  - Diagramy wdrożeniowe

Diagramy te zapewniają uzyskanie wielu perspektyw projektowanego systemu w trakcie jego budowy, celem jej ułatwienia.

Przykładowy diagram klas



UML cieszy się aktualnie dużą popularnością. Prawdopodobnie przez wiele najbliższych lat będzie dominował w obszarze analizy i projektowania.

FAZA PROJEKTOWANIA

Projektowanie → proces transformacji wymagań na postać wykonywalną. Celem fazy projektowania jest udzielenie odpowiedzi na pytanie: Jak system ma być zaimplementowany? Wynikiem jest opis sposobu implementacji.

Uwzględniane szczegóły implementacyjne:

- hardware,
- system operacyjny,
- język programowania,
- biblioteki,
- narzędzia programistyczne.

Rezultaty fazy projektowania

- Poprawiony model
- Uszczegółowiona specyfikacja projektu zawarta w słowniku danych
- Dokument opisujący stworzony projekt składający się z (dla projektowania obiektowego):
  - diagramu klas
  - diagramy sekwencji komunikatów (dla wybranych sytuacji)
  - diagramów stanów
  - zestawień zawierających:
    - definicje klas
    - definicje atrybutów
    - definicje metod
- Zasoby interfejsu użytkownika, np. menu, dialogi
- Projekt bazy danych
- Projekt fizycznej struktury systemu
- Harmonogram fazy implementacji

Rozwój technologii tworzenia oprogramowania:

→ programowanie **sekwencyjne**
(sekwencje + instrukcja skoku)

→ programowanie **strukturalne**
(strukturalne instrukcje: grupujące, pętle, ...)

→ programowanie **proceduralne**
(podprogramy, procedury, funkcje)

→ programowanie **modułowe**
(biblioteki, moduły)

→ programowanie **obiektowe**
(klasy, dziedziczenie, polimorfizm)

→ programowanie . . . . . .

METODY STRUKTURALNE

Wykorzystują:

- Schematy Danych
- Diagramy Przepływu Danych
- Słownik Danych
- Tablice Decyzyjne
- Drzewa Decyzyjne

Reguły podejścia opartego na tworzeniu funkcji:

- Funkcje muszą mieć pojedyncze, zdefiniowane cele.
- Interfejsy do funkcji (wejście i wyjście) powinny jak najprostsze.
- Przy dekompozycji funkcji należy wykorzystywać zasadę „nie więcej niż 7 funkcji podrzędnych”.
- Nazwy funkcji powinny odzwierciedlać ich cel i mówić co ma być zrobione, a nie jak ma być zrobione.

METODY OBIEKTOWE

Obiektowość jest nową ideologią która stara się o uzyskanie jak najmniejszej luki pomiędzy myśleniem o rzeczywistości (dziedzinie problemowej) a myśleniem o danych i procesach, które zachodzą na danych.

Postrzeganie świata rzeczywistego ↔ **Model pojęciowy** ↔ **Schematy struktury danych**

Obiekt składa się z danych oraz z metod działających na tych danych. (obiekt bez danych jest zbiorem funkcji, obiekt bez metod to jest zwykła struktura).

Proces tworzenia modelu obiektowego:

- Identyfikacja klas i obiektów
- Identyfikacja związków pomiędzy klasami
- Identyfikacja i definiowanie pól (atrybutów)
- Identyfikacja i definiowanie metod i komunikatów

Podstawowe zasady obiektowości

- **obiekt** - struktura danych, występująca łącznie z operacjami dozwolonymi do wykonywania na niej, odpowiadająca bytowi wyróżnialnemu w analizowanej rzeczywistości
- **tożsamość** obiektu - wewnętrzny identyfikator (wskaźnik), który pozwala na odróżnienie go od innych obiektów.
- **hermetyzacja** - rozróżnienie pomiędzy interfejsem do obiektu opisującym *co obiekt robi*, a implementacją definiującą, *jak jest zbudowany i jak robi*, to co ma zrobić
- **klasa** - zgrupowanie obiektów o tych samych charakterystykach
- **dziedziczenie** - wielokrotne użycie tego, co wcześniej zostało zrobione: definiowanie klas, które mają wszystkie cechy zdefiniowane wcześniej (z nadklasy) plus cechy nowe
- **polimorfizm** - wybór nazwy dla operacji jest określony wyłącznie semantyką operacji. Decyzja o tym, która metoda implementująca daną operację, zależy od przynależności obiektu do odpowiedniej klasy

Kolejne kroki analizy obiektowej

- Zidentyfikuj klasy i ich atrybuty
- Usuń niepotrzebne klasy i dodaj dziedziczenie
- Wyszukaj ewentualne relacje zawierania się: agregacje, kompozycje.
- Dodaj operacje (metody) do klas poprzez zbudowanie modelu dynamicznego.

Identyfikacja klas - typowe klasy:

- odrębne przedmioty materialne (np. samochód),
- zbiory przedmiotów materialnych (np. marka samochodu),
- zdarzenia istotne dla systemu (np. przegląd gwarancyjny, dostawa),
- role społeczne osób (np. pracownik, wykładowca),
- organizacje (np. serwis samochodowy, firma),
- interakcje między osobami lub systemami (np. pożyczka, spotkanie),
- miejsca przebywania osób lub przedmiotów (np. adres, magazyn), dokumenty (np. faktura, prawo jazdy)

Obiektem jest byt (rzecz lub pojęcie) obserwowalny w świecie rzeczywistym, którego dotyczy rozwiązywany problem.

Obiektem może być także pewien zamknięty fragment oprogramowania (dana, procedura, moduł, dokument, okienko dialogu,...), którym można operować jako zwartą bryłą (wyszukiwać, wiązać, kopiować, blokować, usuwać, indeksować, ...).

Obiekt może być złożony, tj. może składać się z mniejszych obiektów. Klasa może mieć wiele podklas.

Obiekt może być powiązany z innymi obiektami związkami skojarzeniowymi. Występują różne rodzaje związków między klasami np: dziedziczenie, związki logiczne.

Przykład dziedziczenia:

obiekt $\supseteq$ obiekt ruchomy $\supseteq$ pojazd $\supseteq$ pojazd mechaniczny $\supseteq$ auto

Inne związki między klasami:

pracownik naukowy (*pracuje_w*) instytut (*stanowi_część*) wydział (*stanowi_część*) uczelnia (*znajduje_się_w*) miasto . . .

podatnik (*wypełnia*) PIT

adres (*stanowi_część*) danych podatnika

Przykład obiektu

dane:

- Numer konta: 123-4321; Stan konta: 34567 PLN;
  Dane właściciela: Jan Kowalski; Upoważniony: ...; Podpis: …

operacje:

- Załóż konto (inicjalizuj dane)
- Sprawdź (porównaj) podpis
- Sprawdź stan konta
- Wpłać
- Wypłać
- Nalicz procent
- Upoważnij
- Podaj osoby upoważnione
- Zlikwiduj konto

Podsumowanie obiektowości:

- Modelowanie świata rzeczywistego jest ułatwione dzięki zastosowaniu podejścia obiektowego. Klasy, grupujące obiekty świata rzeczywistego, są najbardziej stabilnym elementem dziedziny problemu.
- Hermetyzacja wspomaga redukcję złożoności poprzez zachęcanie użytkowników, by opierali się raczej na interfejsie do obiektu niż jego wewnętrznej organizacji. Abstrahowanie od szczegółów implementacyjnych znacząco ułatwia proces rozumienia.

DODATKOWE SKŁADOWE OPROGRAMOWANIA

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

- składowej interfejsu użytkownika
- składowej zarządzania danymi (przechowywanie trwałych danych)
- składowej zarządzania pamięcią operacyjną
- składowej zarządzania zadaniami (podział czasu procesora)

Do niedawna interfejs użytkownika pochłaniał do 90% kosztów.

Rozwiązaniem jest zastosowanie technik RAD (Rapid Application Development) szybkiego rozwijania aplikacji. Terminem tym określa się narzędzia i techniki programowania umożliwiające szybką budowę prototypów lub gotowych aplikacji, z reguły oparte o programowanie wizualne.

W ostatnich latach nastąpił gwałtowny rozwój narzędzi graficznych służących do tego celu: MS Windows, Object Windows, Microsoft Foundation Class, Visual Component Library, ...

INTERFEJS UŻYTKOWNIKA

Organizacja interakcji z użytkownikiem:

- Za pomocą linii komend → dla niewielkich systemów, dla prototypów, dla zaawansowanych użytkowników.
  Zazwyczaj jest szybszy od interfejsu pełnoekranowego.
- W pełnoekranowym środowisku okienkowym → tworzenie takiego interfejsu ma sens dla dużych systemów. Jest wygodny dla początkujących i średnio zaawansowanych użytkowników

Uwaga na ograniczenia możliwości psychofizycznych użytkownika:
Człowiek może się jednocześnie skupić na 5 - 9 elementach

Różne sposoby wydawania przez użytkownika poleceń systemowi

- Wpisywanie poleceń za pomocą linii komend.
- Wybór opcji z menu.
- Wciśnięcie odpowiedniej kombinacji klawiszy (skrótu).
- Korzystanie z ikon w paskach narzędziowych.
- Wybór przycisku w dialogu.
- Korzystanie z nawigacji kursorem myszy i przycisków myszy

Zasady projektowania interfejsu użytkownika

- **Spójność**. Wygląd oraz obsługa interfejsu powinna być podobna w momencie korzystania z różnych funkcji. Poszczególne programy tworzące system powinny mieć zbliżony interfejs. Taka sama kolorystyka. Umieszczanie pól i przycisków w tych samych miejscach
- **Skróty** dla doświadczonych użytkowników
- **Potwierdzenia** przyjęcia zlecenia użytkownika (dla niebezpiecznych, dla długotrwałych, dla nietypowych operacji)
- Prosta obsługa błędów – czytelne wskazanie przyczyny błędów, możliwość poprawnego wprowadzenia bez potrzeby ponownego wprowadzania innych pól które były poprawne
- Możliwość **odwoływania-cofnięcia** ostatniej (lub kilku ostatnich) akcji.
- **Wrażenie kontroli** nad systemem. Użytkownicy nie lubią, kiedy system sam robi coś, czego użytkownik nie zainicjował, lub kiedy akcja systemu nie daje się przerwać

Określenie fizycznej struktury systemu:

- Określenie **struktury kodu źródłowego**, tj. wyróżnienie plików źródłowych, zależności pomiędzy nimi oraz rozmieszczenie składowych projektu w plikach źródłowych
- Podział systemu na poszczególne aplikacje
- Fizyczne rozmieszczenie danych i aplikacji na stacjach roboczych i serwerach.

Struktura systemu implementowanego w języku C++.