laboratoria 6—8

Modelowanie obiektowej 3-warstwowej architektury

Cel zajęć: Budowa warstwowego modelu architektury tworzonego oprogramowania w postaci diagramu komponentów.
Budowa obiektowego modelu struktury komponentu oprogramowania w postaci diagramu klas.
Zastosowanie wybranych wzorców projektowych zgodnie z zasadami SOLID.
Wstępna implementacja struktury tworzonego oprogramowania na podstawie jego modelu.

Źródła wiedzy: Wykład IO: *Języki graficznego modelowania* (dostępny też w *ePortalu PWr*).
Wykład IO: *Modelowanie struktury diagramami komponentów i diagramami wdrożenia* (dostępny też w *ePortalu PWr*).
Wykład IO: *Modelowanie struktury diagramami klas, diagramami obiektów, diagramami pakietów i diagramami struktur złożonych* (dostępny też w *ePortalu PWr*).
Przykład: *Miniprojekt Biblioteka* (dostępny też w *ePortalu PWr*).
Tutorial *Visual Paradigm*: *How to Draw Component Diagram?*
Tutorial *Visual Paradigm*: *How to Draw Class Diagram?*
Tutorial *Visual Paradigm*: *Instant Generation (of source code from class model)*
Inne: *Wzorce Projektowe (Refactoring Guru)*

Zawartość instrukcji: 4 zadania – ich wykonanie jest oceniane.
2 ćwiczenia pomocnicze.
Spis rzeczy, które należy umieścić w sprawozdaniu z wykonania zadań.
Spis błędów i braków mogących obniżyć ocenę.

Zadania

Zadanie 1

Budowa warstwowego modelu architektury oprogramowania w postaci diagramu komponentów.

Należy utworzyć diagram komponentów w *Visual Paradigm* przedstawiający architekturę tworzonego oprogramowania: jego warstwy i połączenia między nimi. Architektura powinna być 3-warstwowa typu BCE (np. MVC, MVP) lub inna, bardziej złożona.

Najpierw należy określić, z jakich warstw składa się architektura tworzonego oprogramowania i każdą warstwę przedstawić jako osobny komponent o jej nazwie. Nazwa warstwy zależy od wybranej architektury; zwykle są to warstwy: prezentacji (np. *widok*), encji (np. *model*) i kontroli (np. *kontroler*, *prezenter*).

Następnie należy określić połączenia między warstwami, zgodnie z wybraną architekturą, i przedstawić je jako relacje łączące komponenty pośrednio, czyli poprzez klasy graniczne zdefiniowane przy pomocy interfejsów w postaci tzw. *lizaka* i *łapki*, bez używania portów. Połączenia bezpośrednie, np. przy pomocy relacji zależności, są zabronione.

Komponenty powinny być puste, czyli nie powinny zawierać klas. Ich wewnętrzna struktura zostanie wykonania w kolejnych zadaniach jako diagramy klas.

Architektura 3-warstwowa:

Warstwa prezentacji (*widok* w MVC i MVP) – warstwa graniczna, zapewniająca integrację oprogramowania ze światem zewnętrznym, np. GUI użytkownika.

Warstwa encji (*model* w MVC i MVP) – warstwa biznesowa modelująca i przetwarzająca dane oraz łącząca się z magazynem danych, np. bazą danych.

Warstwa kontroli (*kontroler* w MVC, *prezenter* w MVP) – warstwa biznesowa kontrolująca oprogramowanie, zarządzająca procesami biznesowymi oraz pośrednicząca między warstwami prezentacji i zasobów.

Łączenie komponentów poprzez interfejs:

Komponent *A* udostępnia usługę *U* komponentowi *B*, zdefiniowaną w interfejsie *I* w następujący sposób.

Komponent *A* posiada klasę graniczną *KA*, działającą zgodnie ze wzorcem *fasada*, która:

– inicjuje wykonanie usługi *U* na żądanie klasy *KB* komponentu *B*,
– jest dostępna (np. przez referencję) klasie *KB*,

– realizuje interfejs *I*, definiujący operację usługi *U* (nagłówek tej operacji).

Komponent *B* posiada klasę *KB*, która:

– żąda od klasy *KA* komponentu *A* wykonania usługi *U*;

– otrzymuje wynik usługi *U*, jeśli modelująca go operacja zwraca wynik;

– używa interfejsu *I*, czyli zna definicję operacji usługi *U*.

Na diagramie komponentów to połączenie ogranicza się do połączenia interfejsem w postaci tzw. *lizaka* i *łapki* złączonych ze sobą. *Lizak* jest połączony z komponentem zapewniającym usługę (A). *Łapka* jest połączona z komponentem żądającym lub wykorzystującym tę usługę (B).

Zadanie 2

Budowa obiektowego modelu struktury komponentu warstwy kontroli w postaci diagramu klas.

Należy utworzyć diagram klas w *Visual Paradigm* przedstawiający strukturę warstwy kontroli: jej klasy i relacje między nimi, czyli strukturę komponentu tej warstwy, w pakiecie o nazwie tego komponentu.

W tym celu należy wykonać analizę wspólności i zmienności dla scenariuszy przypadków użycia wybranych w poprzednim etapie laboratoriów. Należy przy tym stosować założenia SOLID i odpowiednie wzorce projektowe.

Wymagane wzorce projektowe: *fasada* i *metoda szablonowa* lub *strategia*.

Polecane dodatkowe wzorce projektowe: *łańcuch zobowiązań*.

Na diagramie należy umieścić m.in.:

– główną klasę całego oprogramowania z główną operacją (*main*) i ewentualnie inne klasy i operacje zapewniające działanie oprogramowania składającego się ze wszystkich jego komponentów;

– realizowane i używane interfejsy (klasy ze stereotypem «interface»), które na diagramie komponentów są połączone z tym komponentem;

– fasadowe klasy graniczne, realizujące interfejsy tego komponentu, zbudowane na podstawie wzorca *fasada*;

– inne klasy (zwykłe i abstrakcyjne) wynikające m.in. z zastosowanych wzorców projektowych;

– odpowiednie relacje między klasami: realizację, uogólnienie, asocjację (zwykłą, agregację lub kompozycję) i zależność ze stereotypem «*use*» – używanie klasy lub «*instantiate*» – tworzenie instancji klasy lub ewentualnie bez stereotypu;

– pakiet o nazwie komponentu, zawierający wszystkie klasy tego komponentu, z wyjątkiem interfejsów definiujących usługi innych komponentów, a używanych przez ten komponent; takie interfejsy powinny być na diagramie poza tym pakietem.

Klasy tego komponentu nie mogą być zależne od definicji klas innych komponentów (czyli nie mogą ich wykorzystywać np. jako typ atrybutu czy parametru), z wyjątkiem interfejsów realizowanych przez te inne komponenty.

Interfejs realizowany przez ten komponent, czyli przez jego fasadową klasę graniczną, powinien zawierać operacje inicjujące realizację poszczególnych przypadków użycia.

Warstwa prezentacji nie będzie modelowana diagramem klas ani implementowana, więc komponent warstwy kontroli powinien też symulować działanie warstwy prezentacji, czyli operacje wejścia-wyjścia, w głównej operacji (*main*) lub innym miejscu odpowiedzialnym za operacje realizacji przypadku użycia.

Klasy i operacje odpowiedzialne za realizację przypadków użycia i ich pomocnicze klasy i operacje zostaną dokładniej zamodelowane, dodane nowe i zaimplementowane w następnym etapie laboratoriów. Wtedy ten diagram klas zostanie uzupełniony.

Warstwa kontroli:

Kontroler w architekturze MVC. *Prezenter* w architekturze MVP.

Zapewnia działanie oprogramowania i realizację jego przypadków użycia. Jej klasy sterują działaniem oprogramowania i łączą jego warstwy. Odpowiadają więc za realizację przypadków użycia i przepływ sterowania między klasami tej i innych warstw.

Analiza wspólności i zmienności przypadków użycia:

Ma na celu określenie klas komponentu (tu: warstwy kontroli) i relacji między nimi:

– klasy bazowe: ogólne, abstrakcyjne, interfejsy;

– ich klasy pochodne – powiązane z nimi relacją uogólnienia lub realizacji (odpowiednio do klasy bazowej);

– strukturalne związki między klasami, czyli relacje: asocjacji, agregacji i kompozycji;

– opis tych relacji: kierunek dostępu, liczność, role itd.

– niestrukturalne związki między klasami, czyli relacje zależności;

Założenia SOLID:

S – jedna odpowiedzialność: Klasa lub operacja powinna odpowiadać za tylko 1 zadanie.

O – otwartość / zamkniętość: Klasa lub operacja powinna być otwarta na rozszerzenia i zamknięta na modyfikacje. Jej zachowanie nie powinno wymagać zmiany jej kodu.

L – podstawienie Liskov'ej: Operacja używająca klasę powinna móc używać jej pochodne klasy (po niej dziedziczące) bez ich znajomości.

I – segregacja interfejsów: Interfejsy powinny być podporządkowane zadaniom; nie powinny być ogólne.

D – Odwrócenie zależności: klasa lub operacja powinna zależeć od abstrakcji klasy lub operacji (odpowiednio); nie powinna zależeć od konkretnych implementacji klas lub operacji.

Zastosowanie wzorców projektowych w tym komponencie:

Fasada – graniczna klasa modelowanego komponentu, zapewniająca i ograniczająca dostęp do niego. Klasa innego komponentu, jako klient modelowanego komponentu, wywołuje wybraną operację fasady, żądając danego zachowania modelowanego komponentu. Operacja ta przekazuje sterowanie odpowiednim klasom modelowanego komponentu, w celu obsługi tego żądania. Klasa innego komponentu nie ma dostępu do innych klas modelowanego komponentu.

Metoda szablonowa – dynamiczny wybór sposobu implementacji operacji z kilkuetapowym algorytmem. Algorytm operacji wykonywanej przez modelowany komponent (np. część lub całość realizacji przypadku użycia) składa się z określonych etapów i jest zdefiniowany w abstrakcyjnej klasie. Poszczególne sposoby wykonania etapów tego algorytmu są zaimplementowane w osobnych klasach, dynamicznie wybieranych przez klasę wywołującą tę operację.

Strategia – dynamiczny wybór operacji wykonania danego zadania. Zadanie (np. część lub całość realizacji przypadku użycia) można wykonać na różne sposoby. Poszczególne sposoby są zaimplementowane w osobnych klasach (jako operacja zdefiniowana we wspólnym interfejsie tych klas), dynamicznie wybieranych przez klasę wywołującą tę operację.

Łańcuch zobowiązań – kolejka obiektów różnych klas, wykonująca różnorodne lub złożone zadanie (np. część lub całość realizacji przypadku użycia). Poszczególne obiekty tej kolejki są odpowiedzialne za wykonanie konkretnego zadania lub konkretnej wersji zadania, lub konkretnej części zadania. Żądanie wykonania zadania trafia do początkowego obiektu kolejki. Ten obiekt, jeśli może, to wykonuje to zadanie lub jego część i przekazuje dalej zadanie do wykonania kolejnemu obiektowi w kolejce itd.

Zadanie 3

Budowa obiektowego modelu struktury komponentu warstwy encji w postaci diagramu klas.

Należy utworzyć diagram klas w *Visual Paradigm* przedstawiający strukturę warstwy encji: jej klasy i relacje między nimi, czyli strukturę komponentu tej warstwy, w pakiecie o nazwie tego komponentu.

W tym celu należy wykonać analizę wspólności i zmienności dla scenariuszy przypadków użycia wybranych w poprzednim etapie laboratoriów. Należy przy tym stosować założenia SOLID i odpowiednie wzorce projektowe.

Wymagane wzorce projektowe: *fasada*, *adapter*, *dekorator* oraz *metoda wytwórcza* lub *fabryka abstrakcyjna*.

Polecane dodatkowe wzorce projektowe: *kompozyt* i *obserwator*.

Na diagramie należy umieścić m.in.:

– realizowane i używane interfejsy (klasy ze stereotypem «interface»), które na diagramie komponentów są połączone z tym komponentem;

– fasadowe klasy graniczne, realizujące interfejsy tego komponentu, zbudowane na podstawie wzorca *fasada*;

– inne klasy (zwykłe i abstrakcyjne) wynikające m.in. z zastosowanych wzorców projektowych;

– odpowiednie relacje między klasami: realizację, uogólnienie, asocjację (zwykłą, agregację lub kompozycję) i zależność ze stereotypem «*use*» – używanie klasy lub «*instantiate*» – tworzenie instancji klasy lub ewentualnie bez stereotypu;

– pakiet o nazwie komponentu, zawierający wszystkie klasy tego komponentu, z wyjątkiem interfejsów definiujących usługi innych komponentów, a używanych przez ten komponent; takie interfejsy powinny być na diagramie poza tym pakietem.

Klasy tego komponentu nie mogą być zależne od definicji klas innych komponentów (czyli nie mogą ich wykorzystywać np. jako typ atrybutu czy parametru), z wyjątkiem interfejsów realizowanych przez te inne komponenty.

Interfejs realizowany przez ten komponent, czyli przez jego fasadową klasę graniczną, powinien zawierać operacje inicjujące tworzenie i przetwarzanie danych wejściowych i wyjściowych dla realizacji przypadków użycia.

Warstwa zasobów (np. baza danych) nie będzie modelowana ani implementowana, więc komponent warstwy encji powinien też symulować działanie warstwy zasobów, czyli operacje na bazie danych, w operacjach klasy typu *DAO* (*Data Access Object*) zbudowanej na podstawie wzorca *adapter*.

Klasy i operacje odpowiedzialne za model danych przetwarzanych w realizacji przypadków użycia i ich pomocnicze klasy i operacje zostaną dokładniej zamodelowane, dodane nowe i zaimplementowane w następnym etapie laboratoriów. Wtedy ten diagram klas zostanie uzupełniony.

Warstwa encji:

Model w architekturach MVC i MVP.

Modeluje dane lub struktury danych przetwarzane przez oprogramowanie (np. w celu realizacji przypadku użycia) oraz zapewnia operacje ich przetwarzania. Więc jej klasy określają strukturę i odpowiedzialność poszczególnych obiektów, czyli ich instancji, lub odpowiadają za tworzenie i przetwarzanie tych obiektów na żądanie klas innych warstw.

Zastosowanie wzorców projektowych w tym komponencie:

Fasada – graniczna klasa modelowanego komponentu, zapewniająca i ograniczająca dostęp do niego. Klasa innego komponentu, jako klient modelowanego komponentu, wywołuje wybraną operację fasady, żądając danego zachowania modelowanego komponentu. Operacja ta przekazuje sterowanie odpowiednim klasom modelowanego komponentu, w celu obsługi tego żądania. Klasa innego komponentu nie ma dostępu do innych klas modelowanego komponentu.

Adapter – klasa zapewniająca współpracę klas o niekompatybilnych interfejsach. Klasa kliencka (np. fabryka obiektów: *metoda wytwórcza* lub *fabryka abstrakcyjna*) żąda wykonania operacji innej klasy (np. z warstwy zasobów), której interfejsu nie zna. Zna tylko interfejs adaptera. Adapter zna interfejs tej innej klasy i tłumaczy żądanie pierwszej klasy na żądanie wywołania operacji drugiej klasy.

Szczególnym przykładem adaptera jest *obiekt dostępu do danych* (DAO – *Data Access Object*) – graniczna klasa modelowanego komponentu, komunikująca się z warstwą zasobów (np. z bazą danych). To adapter dla innych klas tego komponentu, które, za jego pośrednictwem, żądają od warstwy zasobów usługi transakcji typu CRUD. DAO tłumaczy ich żądania na odpowiednie operacje warstwy zasobów (np. zapytania do bazy danych).

Dekorator – dynamiczna i rekursywna rozbudowa obiektu (np. przetwarzanych danych) o dodatkowe zachowanie (nowa operacja) lub dodatkowy opis stanu (nowy atrybut), których nie ma definiująca go klasa. Zdefiniowane są w klasie opakowującego ten obiekt dekoratora lub w klasach rekursywnie opakowujących ten obiekt dekoratorów. Klasa kliencka (np. fabryka obiektów: *metoda wytwórcza* lub *fabryka abstrakcyjna*) może utworzyć lub przebudować obiekt nawet tak, że jego zestawu atrybutów i operacji nie można by z góry określić (więc nie ma klasy definiującej takie obiekty).

Metoda wytwórcza – dynamicznie wybierana klasa (fabryka) pojedynczych obiektów. Klasa kliencka, żądająca utworzenia danego obiektu zgodnego ze znanym jej interfejsem, przekazuje to żądanie tej fabryce, która odpowiednio realizuje operację utworzenia tego obiektu. Wyboru fabryki dokonuje na podstawie bieżącej sytuacji. Dana fabryka tworzy obiekt na podstawie tylko jej przypisanej, określającej go klasy.

Fabryka abstrakcyjna – dynamicznie wybierana klasa (fabryka) rodziny kompatybilnych ze sobą obiektów. Klasa żądająca utworzenia danego obiektu zgodnego ze znanym jej interfejsem przekazuje to żądanie tej fabryce, która odpowiednio realizuje operację utworzenia tego obiektu. Wyboru fabryki dokonuje na podstawie bieżącej sytuacji. Dana fabryka tworzy obiekty na podstawie tylko jej przypisanych, kompatybilnie określających je klas.

Kompozyt – drzewiasty kontener obiektów (np. przetwarzanych danych) tej samej lub różnych klas, zapewniający wykonanie tej samej operacji przez wszystkie jego liście, w różny sposób, zdefiniowany przez ich klasy. Klasa żądająca wykonania danej operacji na danych przekazuje to żądanie korzeniowi drzewa, który rekurencyjnie przekazuje je dalej aż do liści, czyli obiektów świadczących żądaną usługę (np. przetworzenie danych, które przechowują). Element drzewa, który jest kontenerem innych elementów drzewa (też kontenerów lub liści) może dodatkowo na swój sposób przetwarzać wyniki operacji wykonanej przez jego liście, liście jego kontenerów itd.

Obserwator – powiadamianie zainteresowanych obiektów (obserwatorów) o zmianie stanu danego (obserwowanego) obiektu. Klasa (np. zarządzająca elementem ekranu użytkownika w warstwie prezentacji) rejestruje się u obserwowanego (np. obiektu przetwarzanych danych w warstwie encji) jako jego obserwator i przekazuje mu dostęp do swojej operacji powiadamiania. Gdy obserwowany zmieni swój stan (np. zmiana wartości atrybutu, wykonanie operacji), to u swoich obserwatorów wywoła operację powiadomienia o tym.

Zadanie 4

Wstępna implementacja struktury oprogramowania na podstawie jego modelu.

Należy wykonać kod wszystkich klas, na podstawie ich modeli wykonanych w zadaniach 2 i 3, w jednym z następujących języków: *Java* (zalecane), *C++*, *C#* lub *Python*.

Kod klasy powinien skupiać się na jej strukturze, czyli przede wszystkim zawierać deklarację klasy oraz jej atrybutów i operacji, uwzględniając związki tej klasy z nią samą i z innymi klasami (uogólnienie, realizacja, asocjacja itd.).

Operacje konieczne do uruchomienia całego oprogramowania, zaczynając od głównej operacji (*main*), powinny być zaimplementowane tak, aby można je było testowo uruchomić w trybie tekstowym.

Operacje odpowiedzialne za realizację przypadków użycia i ich pomocnicze operacje zostaną dokładniej zamodelowane i zaimplementowane w następnym etapie laboratoriów. Teraz, tymczasowo, ich kod powinien być ograniczony do:

– nagłówka operacji (widoczność, nazwa, parametry i typ wyniku);
– zwrócenia domyślnej wartości, w przypadku operacji coś zwracającej;
– lub zgłoszenia wyjątku niezaimplementowania operacji, np. *UnsupportedOperationException()* w języku *Java*
(w miejscu wywołania operacji ten wyjątek powinien być obsłużony).

Idea testowego działania wykonywanego oprogramowania

Uruchomienie oprogramowania zaczyna się od głównej operacji (*main*) będącej w warstwie kontroli, która kolejno:

1) Przygotowuje dane testowe – tworzy obiekty z testowymi wartościami wejściowymi dla operacji przypadków użycia.
2) Przygotowuje klasy i obiekty tworzące poszczególne części oprogramowania – tworzy i inicjuje ich instancje, łączy je ze sobą, ustawia je w stan początkowy itp.
3) Rozpoczyna testowe wykonanie operacji wybranego przypadku użycia (będącego w relacji asocjacji z inicjującym go aktorem) – wywołuje ją w fasadowej klasie warstwy kontroli. Operacja ta jeszcze nie realizuje przypadku użycia; jej tymczasowe zachowanie opisano wyżej w treści zadania.
4) Wyświetla wynik tej operacji – w oknie terminala w trybie tekstowym.
5) Rozpoczyna testowe wykonanie operacji kolejnego wybranego przypadku użycia (jak wyżej) itd.

Wykonanie kodu klas:

Najpierw należy automatycznie wykonać kod klas w *Visual Paradigm* na podstawie diagramów klas. Zaleca się użycie okna *Instant Generator* (menu *Tools* → *Code* → *Instant Generator)* i w nim m.in.:

– wybrać język programowania,
– wybrać katalog docelowy (*Output directory*) na tworzone pliki z kodem,
– zaznaczyć diagramy klas, na podstawie których ma powstać kod.

Następnie należy w wybranym środowisku IDE ręcznie poprawić i uzupełnić kod klas o wyjaśniające go komentarze i dodatkowy kod tak, aby możliwe było uruchomienie i przetestowanie oprogramowania zgodnie z ideą opisaną wyżej.

Zadania 2—4 opracowano częściowo na podstawie instrukcji 5 „Laboratorium z przedmiotu: Inżynieria Oprogramowania W04ITE-SI0011G", autorstwa dr inż. Zofii Kruczkiewicz.

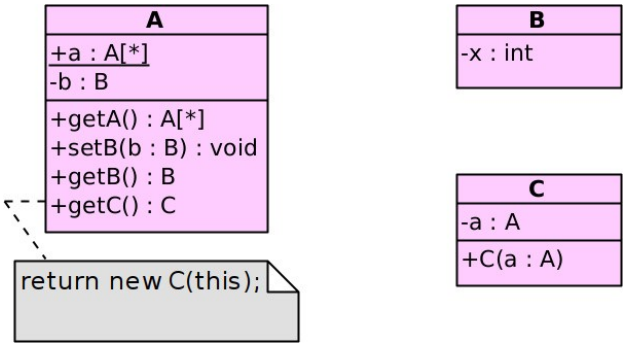
Ćwiczenia

Ćwiczenie 1

Model związków klas na podstawie ich kodu.

Proszę uzupełnić umieszczony niżej diagram klas o związki między nimi i ich pełny opis na podstawie poniższego kodu tych klas w języku *Java*:

```
public class A {
   public static A[] a;
   private B b;
   public A[] getA() {
      return this.a;
   }
   public void setB(B b) {
      this.b = b;
   }
   public B getB() {
      return this.b;
   }
   public C getC() {
      return new C(this);
   }
}
public class B {
   private int x;
}
public class C {
   private A a;
   public C(A a) {
      this.a = a;
   }
}
```



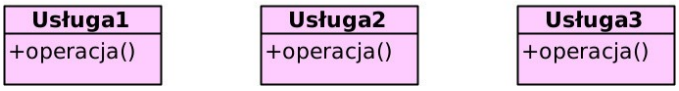
Wskazówka: między tymi klasami są 3 związki strukturalne i 1 związek niestrukturalny.

Ćwiczenie 2
Model dynamicznej zmiany klasy atrybutu.

Proszę uzupełnić umieszczony obok diagram klas na podstawie poniższego opisu:

Klasa *Klient* zawiera atrybut *usługa*, który przechowuje instancję jednej z klas *Usługa1*, *Usługa2* i *Usługa3*.

Każda z tych klas inaczej implementuje operację *operacja()*, z której korzysta *Klient*, dynamicznie (czyli w czasie działania programu) wybrawszy jedną z tych instancji.



Do wykonania ćwiczenia przyda się interfejs lub klasa abstrakcyjna.

Sprawozdanie

Co powinno być na początku sprawozdania:

– Autorzy (imię, nazwisko, nr albumu).
– Nazwa tworzonego oprogramowania (zwięzły opis jego dziedziny).
– Temat tego etapu laboratoriów (*Modelowanie obiektowej 3-warstwowej architektury*).

Co powinno być w treści sprawozdania:

– Diagram komponentów wykonany w zadaniu 1.
– Diagramy klas wykonane w zadaniach 2 i 3.
– Kod klas wykonany w zadaniu 4 (tekst, a nie obraz ze zdjęciem ekranu), zawierający:
  • klasy w kolejności ich w pełni kwalifikowanych nazw (czyli wraz ze ścieżką pakietów, np. *pakiet.klasa.java*),
  • nazwę pliku z definicją klasy (nad tą definicją),
  • całą zawartość tego pliku,
  • numerację linii,
  • komentarze do mało oczywistych rzeczy,
  • wcięcia i inne formatowania ułatwiające czytelność.

Błędy i braki

Jeśli opisane niżej rzeczy mają miejsce, ocena za ten etap laboratoriów może być niższa:

– Diagram komponentów nie określa warstwowej architektury oprogramowania.
– Architektura oprogramowania, określona przez diagram komponentów, jest nieczytelna, trudna do zrozumienia.
– Komponenty na diagramie komponentów łączą się bezpośrednio zamiast przez interfejsy.
– Brak diagramu klas modelującego komponent warstwy biznesowej (warstwy kontroli, encji itp.).
– Na diagramie klas danego komponentu brak interfejsów realizowanych lub używanych przez ten komponent.
– Na diagramie klas danego komponentu brak relacji jego odpowiednich klas z interfejsami realizowanymi lub używanymi przez ten komponent.
– Brak relacji realizacji między klasą realizująca interfejs, a tym interfejsem.
– Klasa używa klasy zdefiniowanej w innym komponencie, niebędącej jej interfejsem lub jej interfejsu realizacją.
– Klasy danego komponentu nie znajdują się (bezpośrednio lub pośrednio) w pakiecie o nazwie tego komponentu.
– Brak klasy z główną operacją (*main*).

– Główna operacja (*main*) znajduje się w innej warstwie niż warstwa kontroli lub w klasie o innej odpowiedzialności niż uruchomienie oprogramowania.
– Brak podkreślenia nazwy atrybutu lub operacji klasy (nie należącej do jej instancji).
– Klasa, której atrybutu typ jest określony przez tę samą klasę lub inną klasę obecną na diagramie (związek strukturalny), jest połączona wskazującą na tę drugą klasę relacją zależności, zamiast odpowiednią relacją asocjacji.
– Klasy połączone związkiem strukturalnym, w którym instancja jednej klasy jest samoistną lub współdzieloną częścią instancji drugiej klasy, są połączone inną relacją niż agregacja.
– Klasy połączone związkiem strukturalnym, w którym instancja jednej klasy jest niesamoistną i wyłączną częścią instancji drugiej klasy, są połączone inną relacją niż kompozycja.
– Węzeł ◊ agregacji lub węzeł ♦ kompozycji jest po stronie klasy posiadanej, zamiast po stronie klasy posiadającej.
– Brak elementu opisu relacji związku strukturalnego (asocjacji, agregacji lub kompozycji), lub jest on umieszczony na jej niewłaściwym końcu: rola, widoczność roli, liczność, grot kierunku dostępu (jeśli ta relacja jest skierowana), kropka na końcu relacji (oznaczająca, że rola podana przy niej jest atrybutem klasy z drugiego końca relacji).
– Brak atrybutu klasy, odpowiadającego roli z drugiego końca jej związku strukturalnego.
– Atrybut klasy, odpowiadający roli z drugiego końca jej związku strukturalnego, jest inny niż ta rola.
– Klasa, która jest strukturalnie związana z tą samą lub inną klasą (przez którąkolwiek relację asocjacji) obecną na diagramie, nie ma atrybutu o typie określonym przez tę drugą klasę.
– Klasa, która nie ma (i ma nie mieć) atrybutu o typie określonym przez tę samą klasę lub inną klasę obecną na diagramie, a jej operacje mogą używać tej drugiej klasy, jest połączona wskazującą na tę drugą klasę którąkolwiek relacją asocjacji, zamiast relacją zależności.
– Typem atrybutu klasy (na diagramie klas), który jest lub może być kolekcją, jest typ tej kolekcji, zamiast typu elementu tej kolekcji.
– Na diagramie klas nie zastosowano czytelnie, poprawnie i praktycznie któregokolwiek z wymaganych wzorców.
– Naruszono bez praktycznego powodu zasady SOLID, np.:
  – klasa ma więcej niż jedną odpowiedzialność;
  – dynamiczne (czyli w czasie działania programu) rozbudowywanie klasy odbywa się przy pomocy dziedziczenia klas, zamiast ich kompozycji/agregacji.
  – klasa dynamicznie zależna od różnych klas (zmiana zależności w czasie działania programu) jest z nimi związana sztywno, zamiast przy pomocy ich abstrakcji (klasy abstrakcyjnej lub interfejsu);
  – podklasa (w relacji realizacji lub uogólnienia) wywołuje operacje swojej nadklasy (naruszenie tzw. reguły *Hollywood*);
– Brak kodu lub fragmentu kodu klasy obecnej na diagramie klas.
– Kod oprogramowania przeczy diagramowi klas.
– Kod klas, w szczególności kod operacji głównej (*main*), uniemożliwia przewidziane treścią zadania testowe uruchomienie oprogramowania.
– Kod nie jest sformatowany lub jest niepraktycznie sformatowany, np. brak komentarzy, wcięć czy numeracji linii.

Ta lista zawiera typowo pojawiające się błędy i braki i nie wyczerpuje powodów, dla których ocena za laboratoria może być niższa.