J.NAWROCKI, M. ANTCZAK, W. FROHMBERG, K. KOLANOWSKI, J. POCHMARA, S. WĄSIK, T. ŻOK

INŻYNIERIA OPROGRAMOWANIA

WPROWADZENIE DO INŻYNIERII OPROGRAMOWANIA

W ogólności cykl życia przedsięwzięcia programistycznego składa się z następujących faz:

a) **zebranie wymagań dotyczących systemu**,
b) **opracowanie projektu systemu** (wykorzystanie języka **UML**),
c) **implementacja systemu w oparciu o zaproponowany projekt rozwiązania**,
d) **sprawdzenie, czy system nie zawiera defektów i czy spełnia wymagania, o które chodziło klientowi**.

Proces zbierania i opracowywania wymagań:

a) ma charakter **cykliczny**,
b) poprzedzony jest **sformułowaniem problemu (lub problemów), które budowany system informatyczny ma rozwiązać i nakreśleniem bardzo ogólnej wizji rozwiązania**,
c) składa się z następujących trzech faz:
   - **zbieranie wymagań** – faza musi uwzględniać, że wymagania często pochodzą od wielu różnych osób i na dodatek należy uwzględniać ograniczenia wynikające np.: z różnych przepisów prawa,
   - **analiza wymagań** – celem tej fazy jest wczesne (najmniej kosztowne) wykrywanie sprzeczności i wzajemnej niejednoznaczności współistniejących wymagań projektowanego systemu,
   - **negocjacja wymagań** – w celu usunięcia zauważonych w powyższej fazie sprzeczności między wymaganiami należy przeprowadzić negocjacje z wieloma zainteresowanymi osobami – osiągnięcie kompromisu często jest trudnym problemem.

Wyróżniamy dwa podstawowe rodzaje wymagań:

a) **funkcjonalne** - opisują funkcje, jakie ma realizować system,
b) **niefunkcjonalne/pozafunkcjonalne** - dotyczą takich aspektów, jak wydajność (np.: szybkość wykonania poszczególnych operacji), czy niezawodność.

Język UML obejmuje wiele różnego typu diagramów, które pozwalają w sposób graficzny **modelować różne aspekty systemów informatycznych (i nie tylko informatycznych)**. Na wykładzie przedstawione zostały bardzo proste przykłady **diagramu stanów** (modelowanie możliwych stanów projektowanego systemu)**, diagramu przypadków użycia** (modelowanie wymagań funkcjonalnych) **i diagramu sekwencji** (ilustrowanie komunikacji między obiektami).

Wyróżniamy dwie podstawowe postaci kontroli jakości:

a) **testowanie** - można wykonywać tylko w odniesieniu do działającego systemu lub jego prototypu,
b) **przeglądy** - można je stosować zarówno do kodu, jak i do specyfikacji wymagań, gdyż ich istotą jest analiza artefaktów. Analiza ta może być przeprowadzona przez pojedynczą osobę (nazywamy to recenzją) lub też przez zespół osób (najpopularniejszym przykładem są inspekcje).

Metody formalne (np.: dobrze znane już sieci Petriego) mają ścisły związek z walidacją oprogramowania. Poprawność programów wykazywana jest na gruncie matematycznym, poprzez **dowodzenie właściwości programów**.

W celu dowodzenia poprawności funkcjonalnej określonej funkcji zapisanej w dowolnym języku programowania np.: C należy:

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

a) związać z tą funkcją **warunek wstępny** (*precondition* – określa relację jakie muszą spełniać parametry wejściowe, aby mogło wystąpić poprawne wykonanie funkcji np.: /*** PRE …… ***/) i **końcowy** (*postcondition* – określa relację, jaka ma być prawdziwa na końcu wykonania podprogramu np.: /*** POST …… ***/),
b) dowieść, że jeżeli warunek wstępny jest spełniony to po wykonaniu mechanizmów zapisanych w podprogramie osiągniemy warunek końcowy. Dowodzenie poprawności programu zawierających pętle odbywa się **metodą niezmienników** (ang. *invariant*). **Niezmiennik** jest to zdanie, które jest prawdziwe za każdym razem, kiedy powtarzane jest wykonanie instrukcji zawartych w pętli. Dokładnie mówiąc, zdanie spełniające powyższą zależność powinno być prawdziwe tuż przed pierwszym wykonaniem instrukcji zawartych w pętli, tuż przed drugim wykonaniem, przed trzecim itd. Zasadniczy problem polega na tym by znaleźć (wymyślić) takie zdanie, który zawsze będzie prawdzie i jednocześnie pomoże nam w udowodnieniu warunku końcowego POST. Powyższa metoda składa się z następujących etapów:
   - **zdefiniowanie i udowodnienie** poprawności odpowiednich niezmienników,
   - **wywiedzenie warunku końcowego** w oparciu o zdefiniowane powyżej niezmienniki.

Systemy zarządzania konfiguracją (np.: CVS, SubVersion), chronią oprogramowanie przed chaotycznymi modyfikacjami i umożliwiają współbieżne modyfikowanie różnych składników oprogramowania w sposób kontrolowany.

Testowanie oprogramowania:

a) jest to **wykonanie kodu** dla kombinacji **danych wejściowych** i **wewnętrznych stanów systemu** w celu **wykrycia błędów**.

Im więcej błędów zostanie wykrytych tym **sprawniejszy** jest **proces testowania**, ale też tym gorzej to świadczy o **procesie tworzenia kodu**. Pracochłonność testowania waha się od **30%-40%** całkowitej pracochłonności w ogólnym przypadku do 70%-80% całkowitej pracochłonności w przypadku **systemów krytycznych**.

Wykonanie przypadku testowego można opisać przy pomocy następujących kroków:

a) testowany system bądź jego fragment są ustawiane w stanie **wstępnym**,
b) podawane są **dane wejściowe** do testowanej implementacji i wyroczni (**wytwarzania wyników oczekiwanych** – najczęściej wyjścia ustalane są przez **projektanta testu**),
c) **zaobserwowane wyjście** jest porównywane z **wyjściem oczekiwanym z wyroczni** w celu ustalenia czy test ujawnił błąd.

Jakość przypadku testowego opisywana jest jako prawdopodobieństwo znalezienia jeszcze nie wykrytego błędu.

Udany test to taki, który wykrywa jeszcze nie wykryty błąd.

SPECYFIKACJA WYMAGAŃ (PRZYPADKI UŻYCIA – USE CASES)

Załóżmy, że jesteśmy analitykami w firmie informatycznej i zbieramy wymagania dla systemu, którego zadaniem będzie informatyzacja działania Księgarni. Podczas wielu spotkań z przedstawicielami uzyskaliśmy zbiór tzw. *„opowieści klienta”* np.:

„Po złożeniu zamówienia przez klienta, dział zamówień pobiera książki z magazynu i przekazuje je do działu wysyłek, zajmującego się przygotowaniem paczki i wysłaniem pod wskazany adres.”

Zaczynamy od identyfikatora przypadku użycia (powiedzmy UC1), oraz odpowiedniej nazwy:

UC1: Obsługa zamówienia

Główny scenariusz:

1. Klient składa zamówienie.

Co się dzieje po złożeniu zamówienia? Dział zamówień powinien sprawdzić to zamówienie. Moglibyśmy napisać:

2. Dział zamówień sprawdza dostępność zamówionych książek w magazynie.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

Lecz zawsze zapisujemy akcje, zakładając powodzenie jej wykonania, czyli że w magazynie książki są (bez konstrukcji warunkowych):

3. Dział zamówień pobiera książki z magazynu.

A co jeżeli książek tam nie będzie? Tym się zajmiemy w drugiej kolejności – najpierw należy skończyć główny scenariusz:

4. Po skompletowaniu książek dział zamówień przekazuje je do działu wysyłek wraz z adresem klienta.
5. Dział wysyłek pakuje książki i wysyła je do klienta.

Dopiero po skończeniu scenariusza głównego powinniśmy się zająć sytuacjami wyjątkowymi. Wcześniej jednak należy przedstawić je osobie, która zna się na dziedzinie problemu (np.: użytkownik końcowy, klient), aby sprawdzić czy jednakowo (się) je zrozumieliście. Spójrzmy na nasz scenariusz i spróbujmy znaleźć akcje, które mogą się nie powieść:

W kroku 1. Klient w trakcie składania zamówienia może zrezygnować.

W kroku 3. W magazynie może już nie być pewnej książki. Zapisujemy (więc) warunki dla tych sytuacji. Każdą taką sytuację oznaczamy kolejną literą alfabetu (dla każdego kroku):

Rozszerzenia:

1a. Klient zrezygnował ze składania zamówienia.

3a. Niektórych książek w magazynie nie było.

To jest trzeci etap – wyszukiwanie warunków niepowodzenia. Następnie dla każdego takiego warunku musimy przygotować akcje alternatywne:

3a1. Magazynier zamawia brakujące książki z wydawnictwa.

3a1a. Nakład niektórych książek się wyczerpał – nie można ich już kupić.

3a1a1. Magazynier przekazuje tą informację do działu zamówień.

3a1a2. Dział zamówień informuje klienta o niemożliwości zrealizowania zamówienia.

3a2. Dział zamówień czeka z wysłaniem zamówienia (tak długo,) dopóki książki się nie pojawią.

PROJEKTOWANIE SYSTEMU – JĘZYK UML

Opisuje ona system na wysokim poziomie abstrakcji. Pomaga w specyfikacji, wizualizacji i dokumentacji modeli systemów włączając w to ich strukturę i projekt, w taki sposób, iż spełniają one wszystkie wymagania. Służy do modelowania dziedziny problemu przy wykorzystaniu reprezentacji graficznej – diagramów. W starszej wersji UML wyróżniano tylko dziewięć rodzajów diagramów. Obecnie w najnowszej wersji wyróżnia się trzynaście rodzajów diagramów. UML wspomaga specyfikowanie wszystkich ważnych decyzji analitycznych, projektowych i implementacyjnych, które muszą być podejmowane podczas wytwarzania i wdrażania systemu informatycznego. Modele w nim zapisane mogą być wprost powiązane z wieloma językami programowania. To połączenie umożliwia tzw. inżynierię do przodu, czyli możliwość wygenerowania kodu w języku programowania na podstawie modelu UML. Możliwe jest także odwrotne przekształcenie, tzw. inżynieria wstecz, czyli stworzenie modelu na podstawie kodu.

Diagramy te są podzielone na cztery rodzaje diagramów abstrakcyjnych:

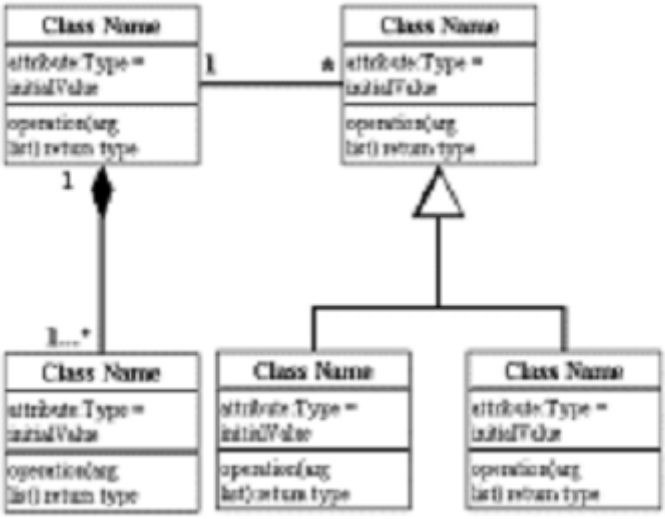
1. struktur,
2. dynamiki,
3. wdrożeniowe,
4. interakcji.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

Pierwsze sześć diagramów (1 – 6) należy do diagramów struktur. Kolejne trzy diagramy (8 – 10) należą do diagramów dynamiki. Diagram wdrożeniowy (7) należy oczywiście do diagramów wdrożeniowych. Z kolei ostatnie trzy diagramy (11 – 13) należą do diagramów interakcji.

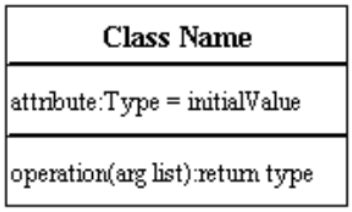
DIAGRAM KLAS

Jest podstawą prawie każdej metody zorientowanej obiektowo. Opisuje statyczną budowę systemu.



Klasy przedstawiają abstrakcje jednostek ze wspólnymi cechami charakterystycznymi. Asocjacje przedstawiają związki pomiędzy klasami. Poniżej przedstawione zostały podstawowe elementy, które składają się na diagram klas.

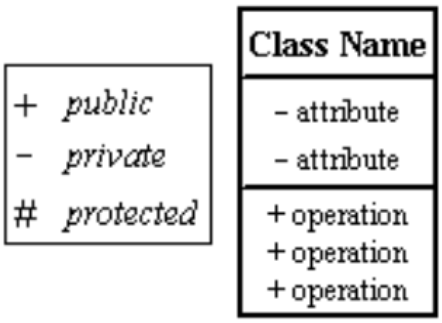
Klasy – przedstawione jako prostokąty podzielone na trzy części; zawierają nazwę klasy w pierwszym pod2prostokącie, listę atrybutów w drugim oraz operacje w trzecim.



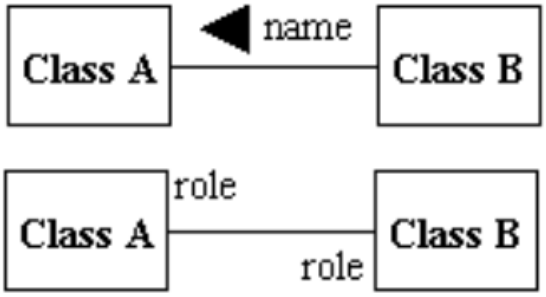
Widoczność – określa poziom dostępu do informacji zawartej w danej klasie;

- *private* ukrywa informacje przed innymi klasami z zewnątrz,
- *public* pozwala innym klasom widzieć informacje danej klasy,
- *protected* pozwala dzieciom (klasom dziedziczącym) danej klasy na dostęp do informacji, a ukrywa je przed klasami zewnętrznymi.

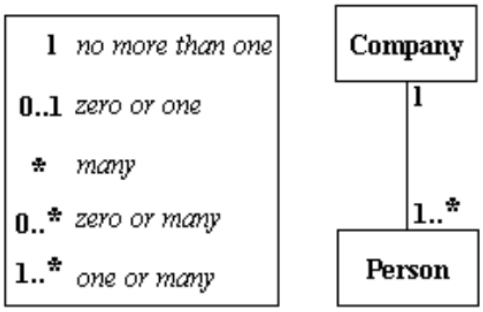
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.



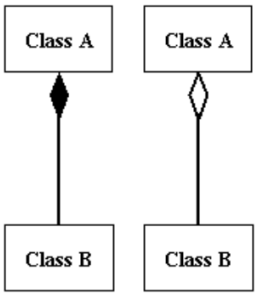
Asocjacje – reprezentują statyczne zależności pomiędzy klasami; posiadają nazwę nad lub pod symbolem (linią) oraz zawierają strzałkę określającą kierunek zależności; na swoich końcach posiadają role reprezentujące, w jaki sposób dwie klasy widzą się nawzajem; rzadko wykorzystuje się jednocześnie nazwę asocjacji oraz role.



Liczność – umieszczane są na końcach asocjacji; określają liczbę instancji klasy połączonej z instancją innej klasy.



Kompozycja i agregacja – kompozycja jest specjalnym typem agregacji, który oznacza silną zależność między całą klasą A, a częścią klasy B; oznaczana jest przez wypełniony romb; pusty romb określa prostą agregację, w której *„cała”* klasa odgrywa ważniejszą rolą niż jej *„część”*, ale dwie klasy nie są od siebie zależne; w obu przypadkach romb znajduje się po stronie *„całej”* klasy.



Generalizacja – jest inną nazwą dziedziczenia; odnosi się do zależności pomiędzy dwoma klasami gdzie jedna jest wyspecjalizowaną wersją drugiej.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

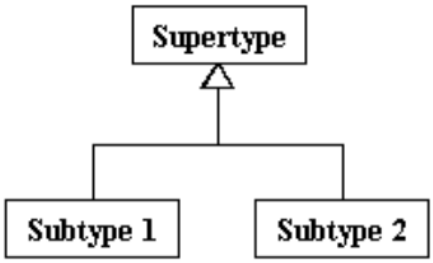


DIAGRAM OBIEKTÓW

Opisują statyczną strukturę systemu w konkretnym czasie. Mogą być wykorzystane do testowania dokładności diagramów klas.

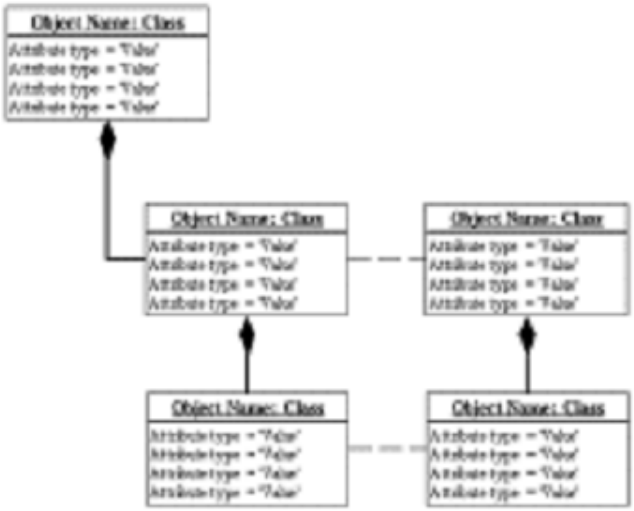
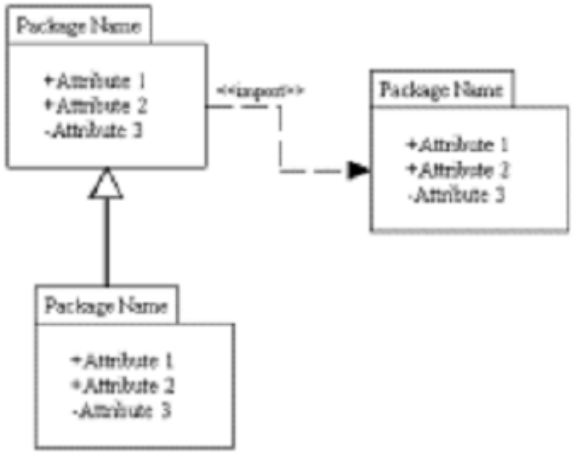


DIAGRAM PAKIETÓW

Są podzbiorem diagramów klas, choć są czasami traktowane jako osobna technika modelowania. Organizują elementy systemu w powiązane grupy w celu zminimalizowania zależności pomiędzy pakietami.

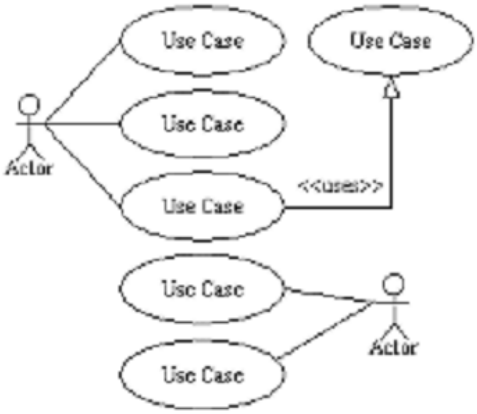


(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

DIAGRAM PRZYPADKÓW UŻYCIA

Modelują funkcjonalność systemu używając do tego aktorów oraz przypadki użycia.
Pomiędzy przypadkami użycia mogą zachodzić następujące relacje:

- *include/uses* – oznacza, że dany przypadek użycia zawiera inny,
- *extend* – oznacza, że dany przypadek użycia może być opcjonalnie wykonany w ramach innego.



Przykład:

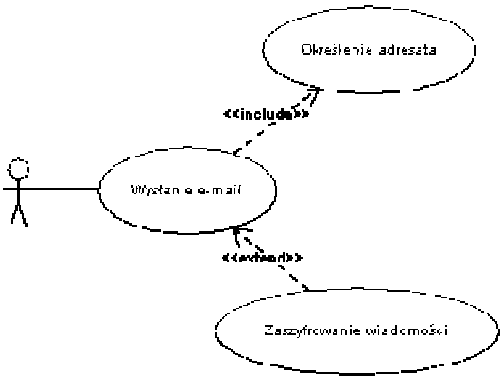
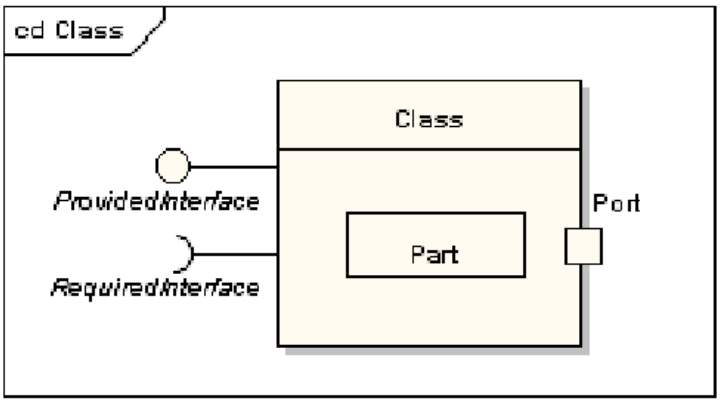


DIAGRAM STRUKTUR ZŁOŻONYCH

Przedstawia wewnętrzną strukturę klasy łącznie z punktami interakcji do innych części systemu. Pokazuje konfigurację oraz związki części, które razem przedstawiają zachowanie zawartej klasy. Elementy klasy są opisane bardzo szczegółowo w części klasy diagramu. Sekcja ta opisuje sposób, w jaki klasy mogą być przedstawione jako złożone elementy wystawiające interfejsy oraz zawierające porty i części.



(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

DIAGRAM KOMPONENTÓW

Opisuje organizację fizyczną komponentów programu włączając w to kod źródłowy, kod binarny oraz pliki wykonywalne.

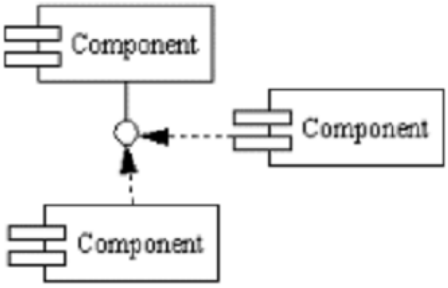


DIAGRAM WDROŻENIOWY

Przedstawia fizyczne zasoby systemu takie jak węzły, komponenty oraz połączenia.

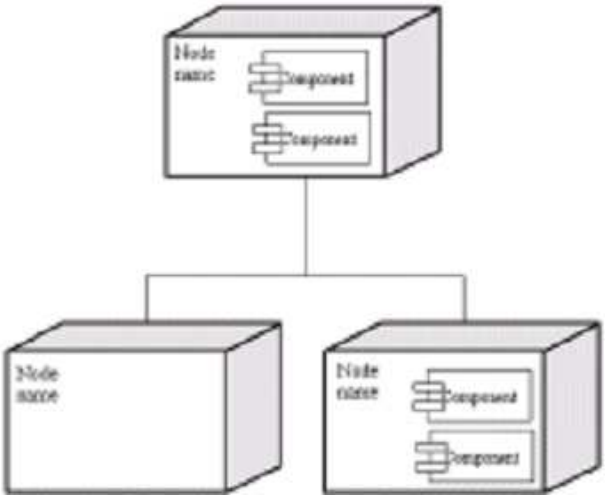


DIAGRAM CZYNNOŚCI

Przedstawiają dynamiczną strukturę systemu poprzez modelowanie przepływu kontroli od czynności do czynności. Czynność reprezentuje operację wykonywaną na jakiejś klasie systemu, której wynikiem jest zmiana stanu systemu. Zazwyczaj diagramy te są wykorzystywane do modelowania przepływów, procesów biznesowych oraz operacji wewnętrznych.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

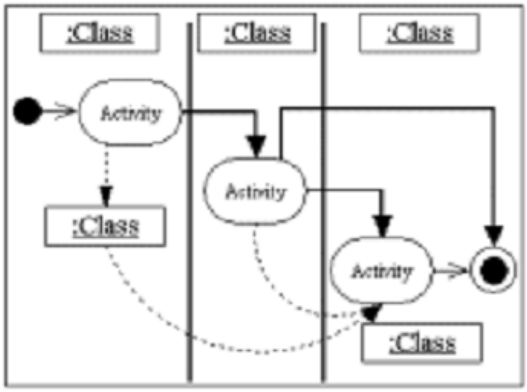


DIAGRAM STANÓW (MASZYNY STANOWEJ)

Opisuje dynamiczne zachowanie systemu w odpowiedzi na zewnętrzną stymulację. Są szczególnie przydatne podczas modelowania obiektów, których stany są wyzwalane przez specyficzne zdarzenia

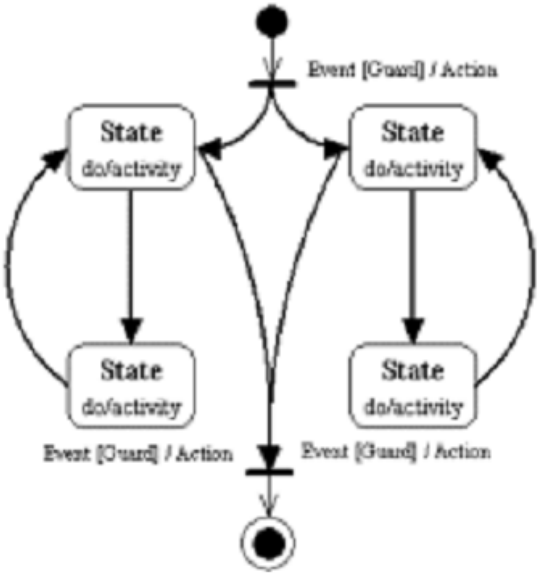


DIAGRAM ZALEŻNOŚCI CZASOWYCH

Wykorzystywane są do przedstawiania zmian stanu lub wartości jednego lub więcej elementów w czasie. Mogą również pokazywać interakcje pomiędzy zdarzeniami czasowymi a czasem oraz ograniczeniami ich czasu trwania.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

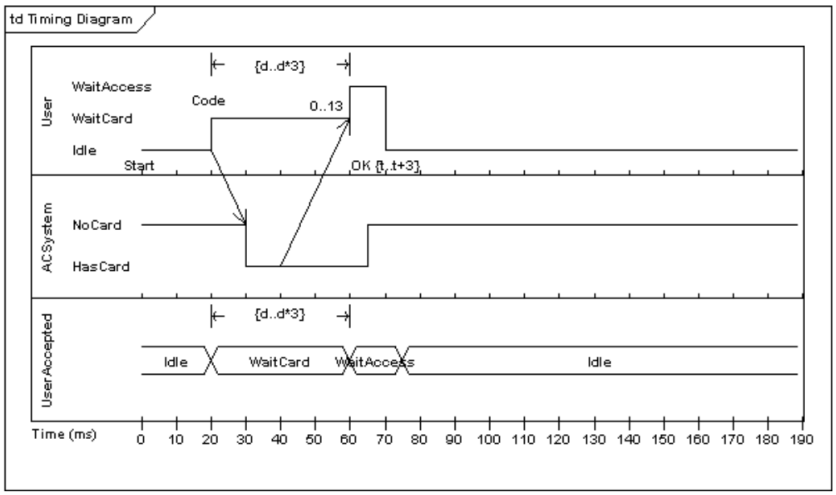
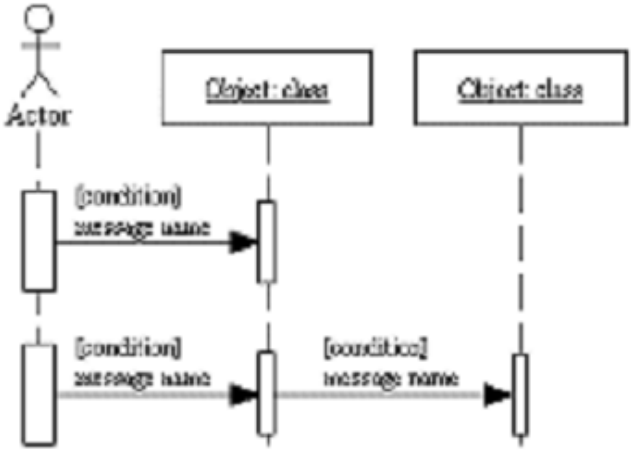


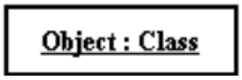
DIAGRAM SEKWENCJI

Opisują interakcje pomiędzy klasami w znaczeniu wymiany wiadomości w czasie.

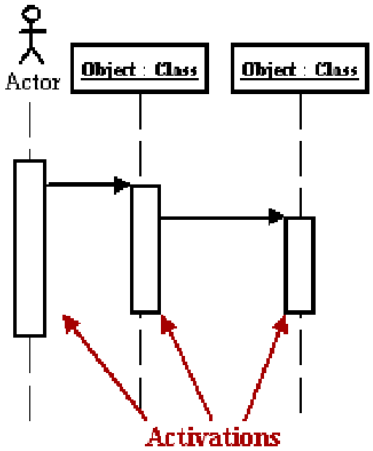


Poniżej przedstawione zostały podstawowe elementy, które składają się na diagram sekwencji.

Rola klasy – opisuje sposób, w jaki obiekt będzie zachowywał się w danym kontekście. Nie zawiera listy atrybutów obiektu.

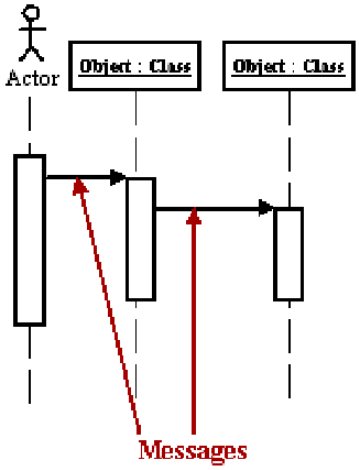


Aktywacja – prostokąty reprezentujące czas, jaki obiekt potrzebuje do zakończenia zadania.



Wiadomości – są to strzałki, które reprezentują komunikację pomiędzy obiektami.

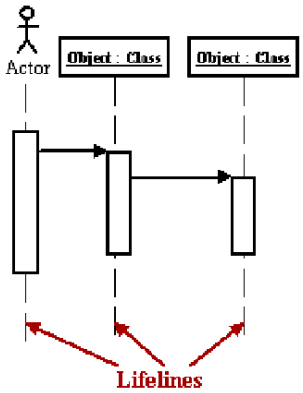
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.



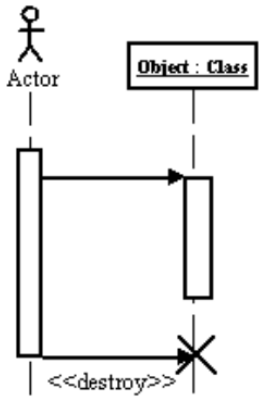
Wiadomości asynchroniczne są wysyłane z obiektu, który nie czeka na odpowiedź od odbiorcy wiadomości przed kontynuowaniem swoich zadań.

Arrow	Message type
→	Simple
→	Synchronous
→	Asynchronous
↺	Balking
🕒 →	Time out

Linie życia – są to pionowe linie, które wskazują na obecność obiektu w czasie.

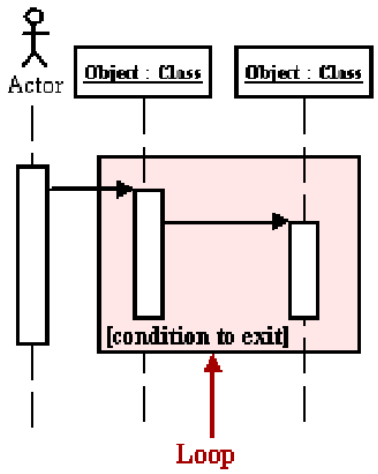


Zniszczenie obiektów – obiekty mogą być zakończone wcześniej poprzez użycie strzałki oznaczonej przez *„<<destroy>>”*, która wskazuje na X.



Pętle – oznaczane są poprzez prostokąt. W kwadratowych nawiasach w dolnym lewym narożniku umieszczany jest warunek istnienia pętli.

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.



Przykładowy diagram dla UC1

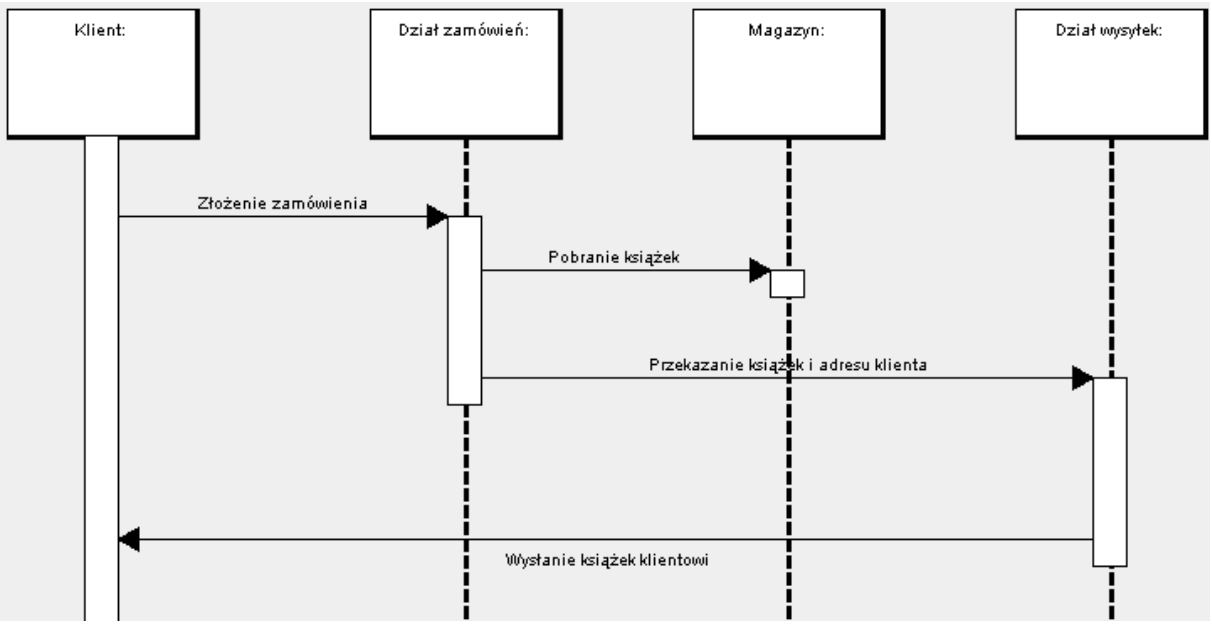
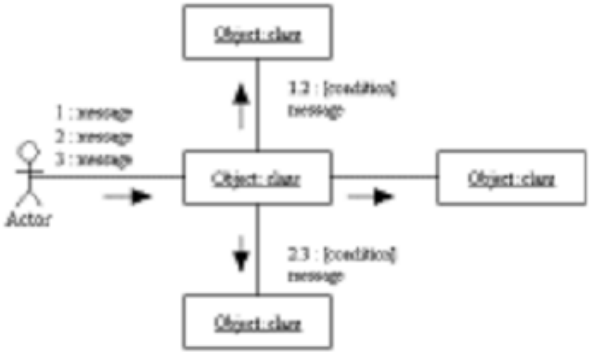


DIAGRAM KOMUNIKACJI (KOLABORACJI)

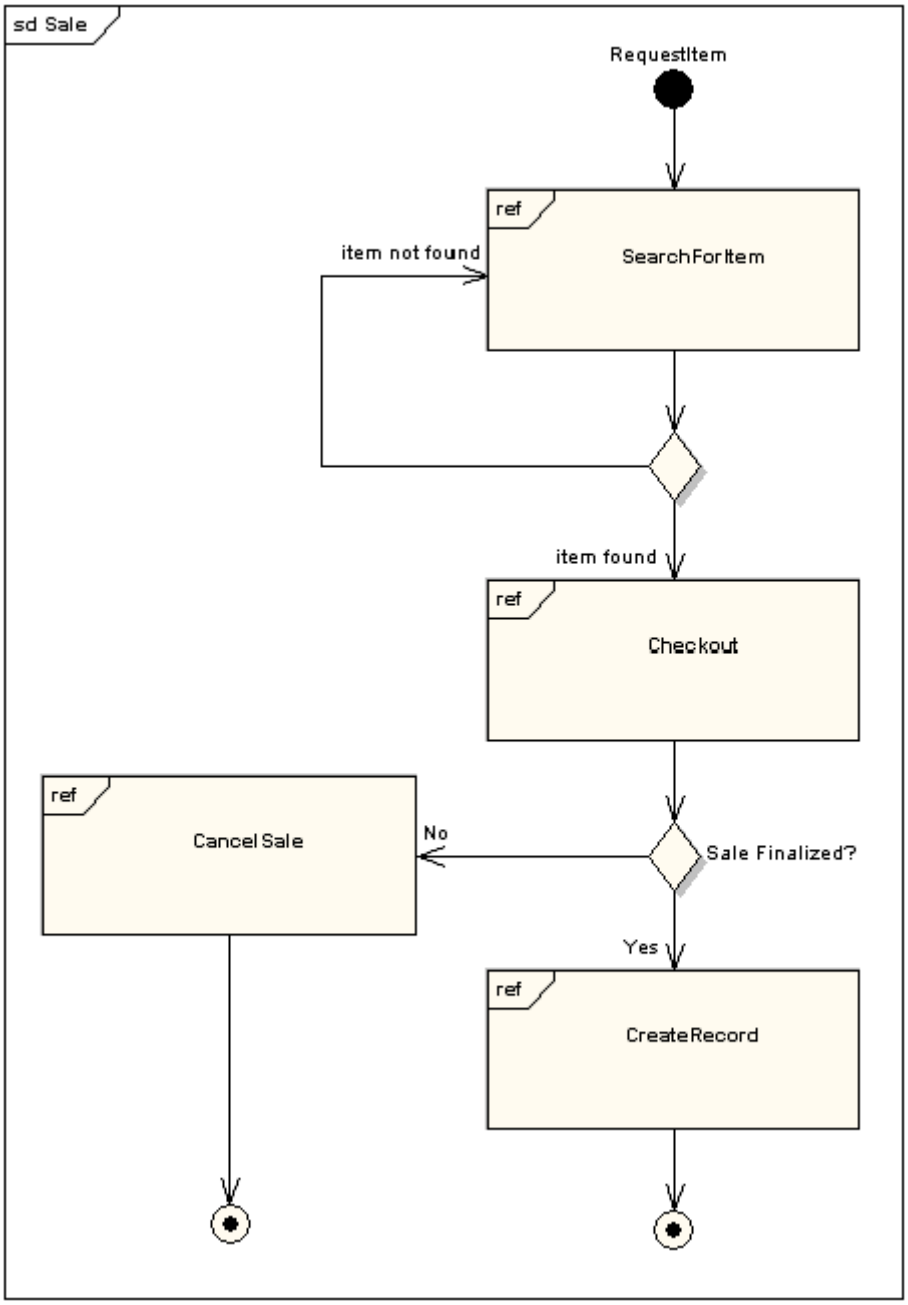
Reprezentuje interakcje pomiędzy obiektami jako serie kolejnych wiadomości. Opisują zarówno statyczną strukturę jak i dynamiczne zachowanie systemu.



(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

DIAGRAM STEROWANIA INTERAKCJĄ

Jest on szczególnym przypadkiem diagramu czynności, w którym węzły reprezentują diagramy interakcji. Diagramy sterowania interakcją mogą zawierać sekwencje, komunikacje, opis interakcji oraz diagramy zależności czasowych. Większość notacji dla tych diagramów jest taka sama jak dla diagramów czynności. Jednakże wprowadzają one dwa nowe elementy: interakcję występowania oraz elementy interakcji.



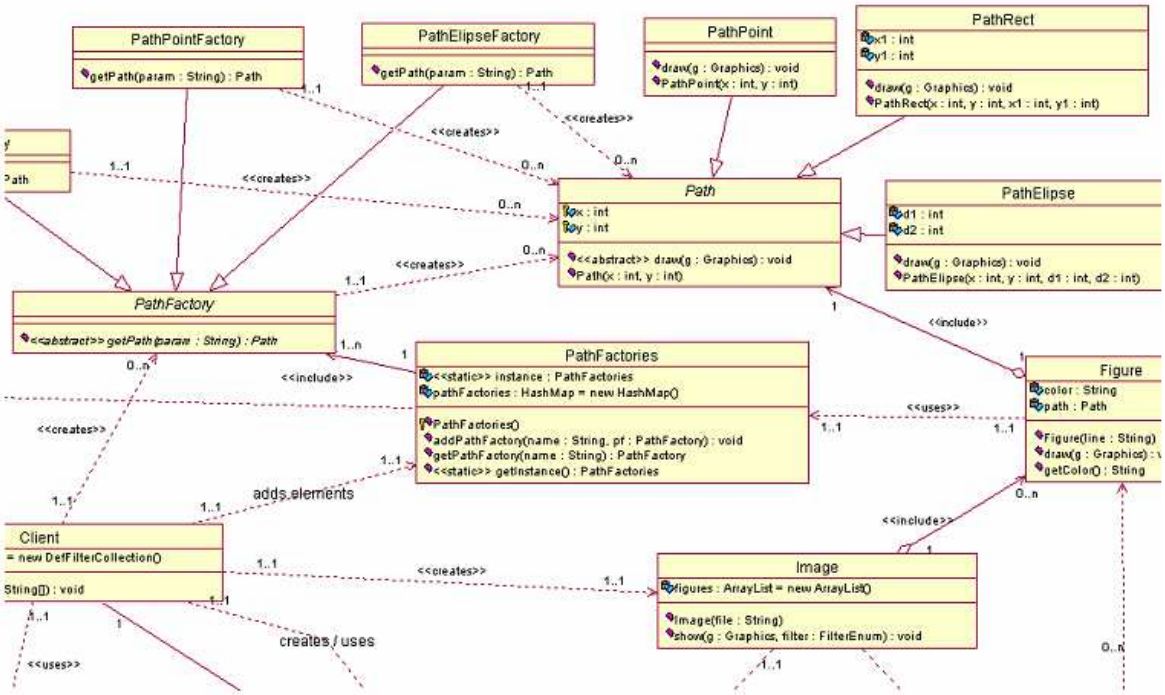
PRZYKŁADOWY PROJEKT I JEGO ROZWIĄZANIE

Zaprojektować filtr, który z pliku wejściowego o ustalonej budowie odczytuje obrazek. Obrazki są wektorowe – zbiory figur płaskich (elipsa, prostokąt, punkt). Każda figura ma kolor. Zadaniem filtra jest

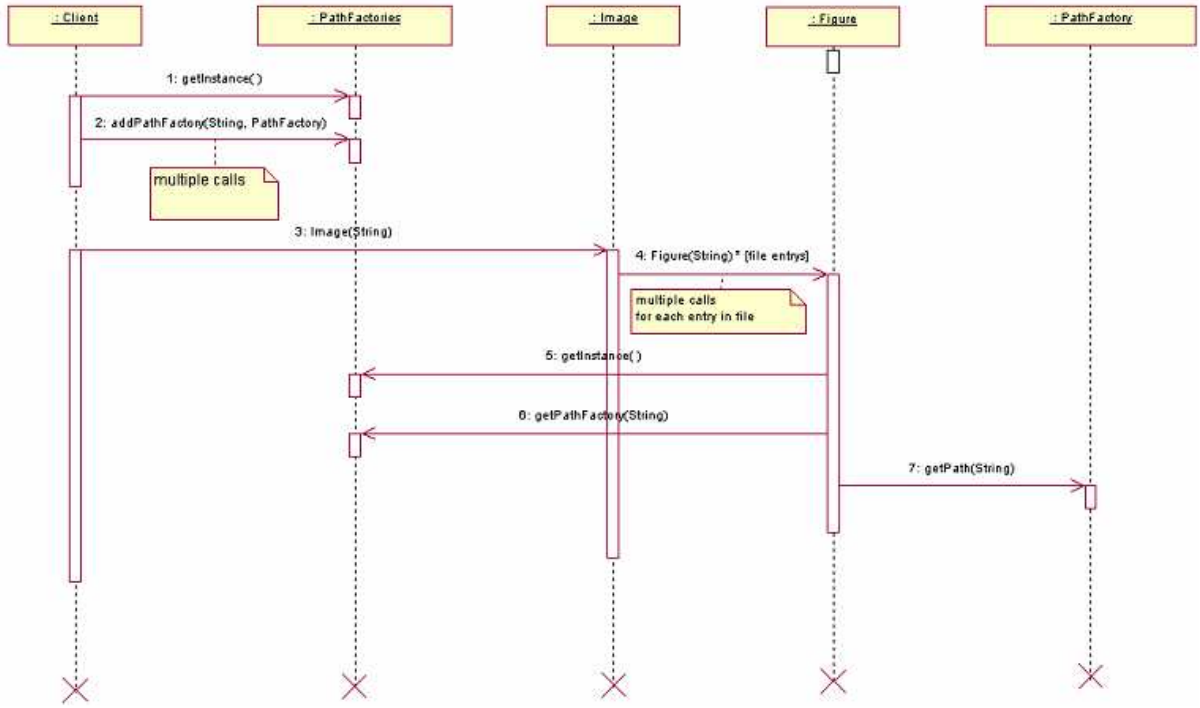
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

wyświetlenie tylko tych figur, które maja określony kolor. Liczba instancji filtra może być większa od 1. Jeden filtr oznacza filtrowania jednego koloru, czyli jeśli maja być filtrowane dwa kolory to dwie instancje filtrów. Obrazki przechodzą przez wiele filtrów. Zakłada się, że z czasem liczba filtrów może zostać powiększona o nowe.

Przykładowy *„wycinek”* diagramu klas



Przykładowy diagram sekwencji



ArgoUML jest jednym z najlepszych narzędzi *opensource* wspomagających modelowanie systemów z wykorzystaniem języka UML (`http://argouml.tigris.org/`).

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

AUTOMATYZACJA TESTÓW JEDNOSTKOWYCH (*CUNIT*) ORAZ GENERACJA DOKUMENTACJI UŻYTKOWEJ *(DOXYGEN)*

Naszym zadaniem jest napisanie prostego programu w języku C (kalkulator) realizującego proste funkcje dodawania i mnożenia liczb całkowitych oraz zaproponowanie i zaimplementowanie kilku przypadków testowych, wspomagających wykrywanie błędów w pisanym przez nas oprogramowaniu.
W celu szybkiej konfiguracji środowiska uruchomieniowego należy pobrać pakiet *„CUnitPowerPack.zip”*, który zawiera środowisko programistyczne *Dev-C++*, bibliotekę *CUnit*, podręcznik wspomagający (krok po kroku) instalację i konfigurację powyższych narzędzi oraz projekt *CUExample*, który jest przykładową implementacją powyższego zadania i może być wykorzystywany w celu zrozumienia idei i implementacji testów jednostkowych.
Przykładowy projekt *CUExample,* stworzony w środowisku Dev-C++, składa się z następujących plików:

- src/main.c

```
#include "includes/calc.h"

int main() {
   int a,b;
   printf("Define a:");
   scanf("%d",&a);
   printf("Define b:");
   scanf("%d",&b);
   printf("\nAdding result: %d",add(a,b));
   printf("\nSubtracting result: %d",sub(a,b));
   printf("\nMultiplying result: %d\n\n",mul(a,b));
   system("PAUSE");
   return 0;
}
```

- src/calc/calc.c

```
#include "../includes/calc.h"

int add(int a, int b) {
    return a+b;
}

int sub(int a, int b) {
    return a-b;
}

int mul(int a, int b) {
    return a*b;
}
```

- src/includes/calc.h

```
/*! \file calc.h
    \brief Header file for calc module.
*/

#ifndef _CALC_H_
#define _CALC_H_

#include <stdio.h>
#include <stdlib.h>

/*! \fn int add(int a, int b);
    \brief Adding two values.
    \param a First value.
    \param b Second value.
*/
```

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

```
int add(int a, int b);

/*! \fn int sub(int a, int b);
    \brief Subtracting two values.
    \param a First value.
    \param b Second value.
*/
int sub(int a, int b);


/*! \fn int mul(int a, int b);
    \brief Multiplying two values.
    \param a First value.
    \param b Second value.
*/
int mul(int a, int b);

#endif
```

- test/mainTest.c

```
#include <stdio.h>
#include <string.h>
#include "CUnit/Automated.h"
#include "../src/includes/calc.h"
/* The suite initialization function.
 * Returns zero on success, non-zero otherwise.
 */
int init_suite(void)
{
   return 0;
}

/* The suite cleanup function.
 * Returns zero on success, non-zero otherwise.
 */
int clean_suite(void)
{
   return 0;
}

void test_calc_add_true(void)
{
      CU_ASSERT(3 == add(1,2));
}

void test_calc_add_false(void)
{
      CU_ASSERT(4 != add(1,2));
}

void test_calc_sub_true(void)
{
      CU_ASSERT(-1 == sub(1,2));
}

void test_calc_sub_false(void)
{
      CU_ASSERT(2 != sub(1,2));
}
```

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

```
void test_calc_mul_true(void)
{
      CU_ASSERT(2 == mul(1,2));
}

void test_calc_mul_false(void)
{
      CU_ASSERT(3 != mul(1,2));
}

/* The main() function for setting up and running the tests.
 * Returns a CUE_SUCCESS on successful running, another
 * CUnit error code on failure.
 */
int main()
{
   CU_pSuite pCalcSuite = NULL;

   /* initialize the CUnit test registry */
   if (CUE_SUCCESS != CU_initialize_registry())
      return CU_get_error();

   /* add a suite to the registry */
   pCalcSuite = CU_add_suite("calc_suite", init_suite, clean_suite);
   if (NULL == pCalcSuite) {
      CU_cleanup_registry();
      return CU_get_error();
   }

   /* add the tests to the suite */
  if ((NULL == CU_add_test(pCalcSuite, "calc_add_true", test_calc_add_true)) ||
      (NULL == CU_add_test(pCalcSuite, "calc_add_false", test_calc_add_false))||
      (NULL == CU_add_test(pCalcSuite, "calc_sub_true", test_calc_sub_true))||
      (NULL == CU_add_test(pCalcSuite, "calc_sub_false", test_calc_sub_false))||
      (NULL == CU_add_test(pCalcSuite, "calc_mul_true", test_calc_mul_true))||
      (NULL == CU_add_test(pCalcSuite, "calc_mul_false", test_calc_mul_false)))
  {
      CU_cleanup_registry();
      return CU_get_error();
   }

   /* Run all tests using the CUnit Automated interface */
   CU_set_output_filename("CUExampleTests");
   CU_list_tests_to_file();
   CU_automated_run_tests();
   CU_cleanup_registry();
   return CU_get_error();
}
```

(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.