J.NAWROCKI, M. ANTCZAK, H. ĆWIEK, W. FROHMBERG, A. HOFFA, M. KIERZYNKA, S. WĄSIK

INŻYNIERIA OPROGRAMOWANIA

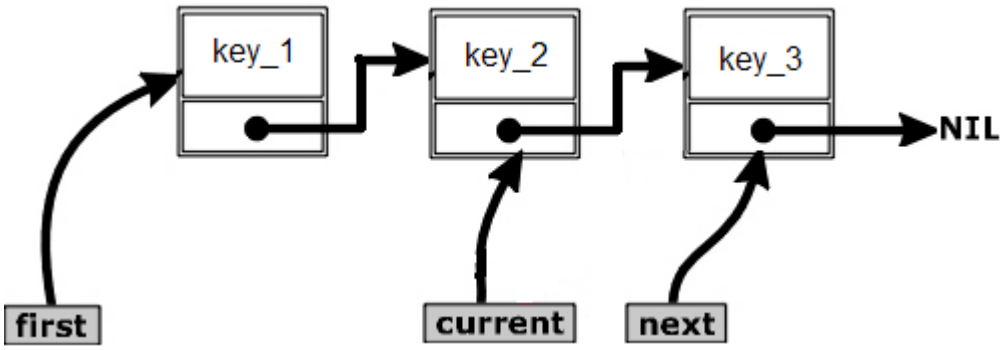
STRUKTURY DANYCH

Wyróżniamy następujące dynamiczne struktury danych:

LISTA

Lista jest liniowo uporządkowaną strukturą zawierającą zbiór elementów, z których dowolny element można usunąć oraz dodać w dowolnym miejscu. Pierwszy i ostatni element listy nazywamy *końcami listy*. Każdy element ma co najwyżej jednego następnika, który składa się z następujących składowych: klucza identyfikacyjnego, wskaźnika do następnika oraz dodatkowo dalszych informacji. Listy mogą być posortowane w porządku niemalejącym (w korzeniu znajduje się element o najmniejszej wartości klucza) lub nierosnącym (w korzeniu znajduje się element o największej wartości klucza). Rozważmy przypadek **jednokierunkowej listy posortowanej** w porządku niemalejącym. Aby dodać nowy element należy sprawdzić, w którym miejscu powinien się on znajdować. Dokonywane jest odpowiednie sprawdzenie rozpoczynające się w korzeniu i przechodzenie przez kolejne elementy w liście, tak długo dopóki nowy element, który chcemy dodać jest większy od badanego węzła i mniejszy od jego następnika, wtedy należy umieścić go między nimi. Należy, więc ustawić wskaźnik aktualnego węzła na dodawany element, a wskaźnik tego elementu na następnik. Ponieważ jest to lista jednokierunkowa, przeszukiwanie jej należy zawsze zaczynać od korzenia. Dodając, więc pierwszy element do pustej listy należy zapamiętać jego wskaźnik, by później mieć dostęp do tej struktury dzięki niemu. Listy mogą zawierać dwa wskaźniki. Takie listy nazywamy **dwukierunkowymi,** które pozwalają na poruszanie się w niej w obydwu kierunkach, co przyspiesza wszystkie operacje.

- *Jednokierunkowa*

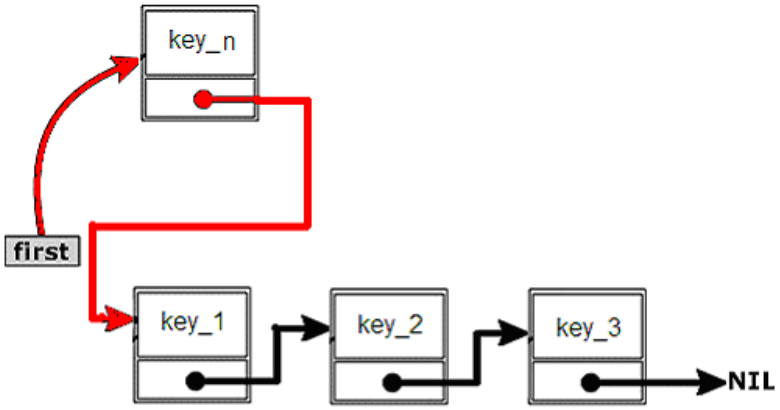


(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

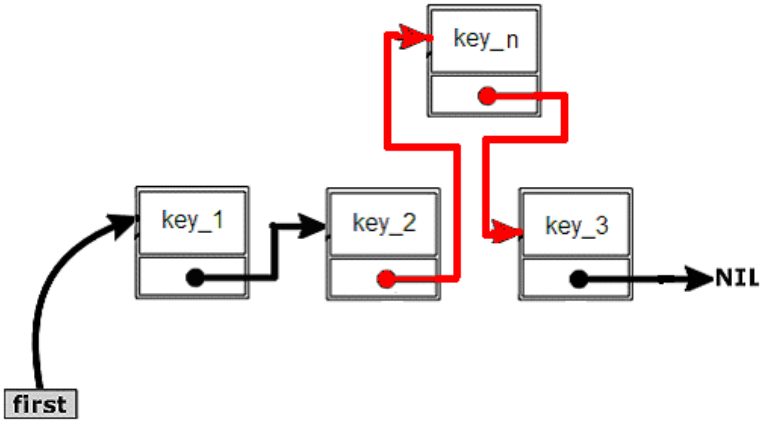
Przeszukiwanie dokonuje się od pierwszego elementu na liście (wskazywanego przez wskaźnik *first*) do ostatniego, który w polu swojego następnika wskazuje na wartość pustą (*NIL*).

DODAWANIE NOWYCH ELEMENTÓW

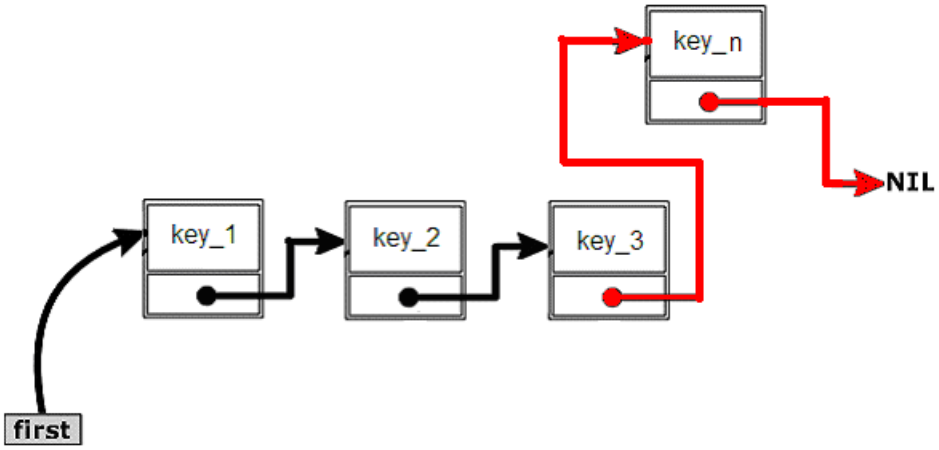
a) na początku



b) w środku (między 2 a 3 elementem)



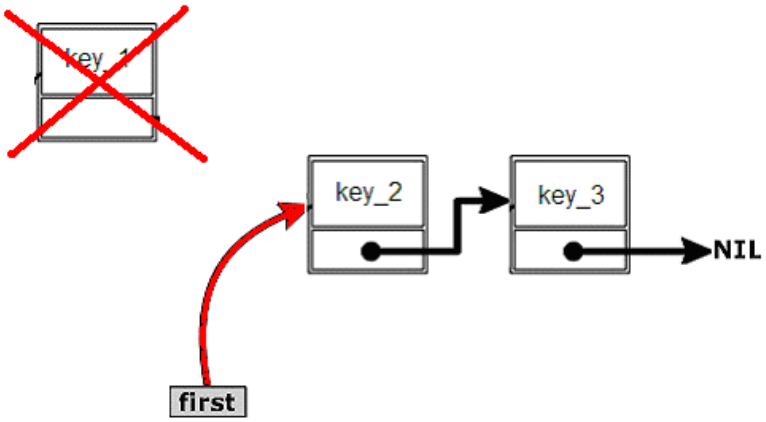
c) na końcu



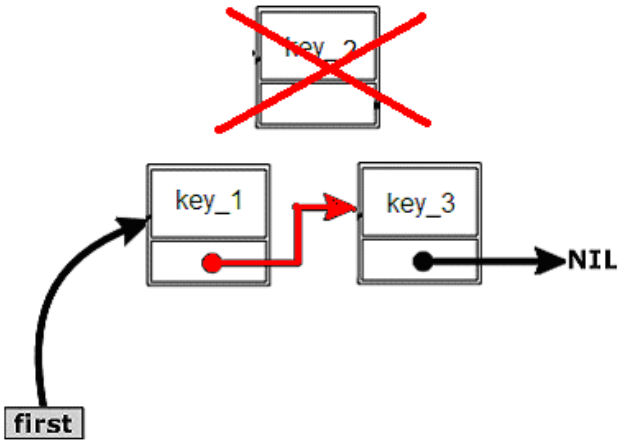
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

USUWANIE ISTNIEJĄCYCH ELEMENTÓW:

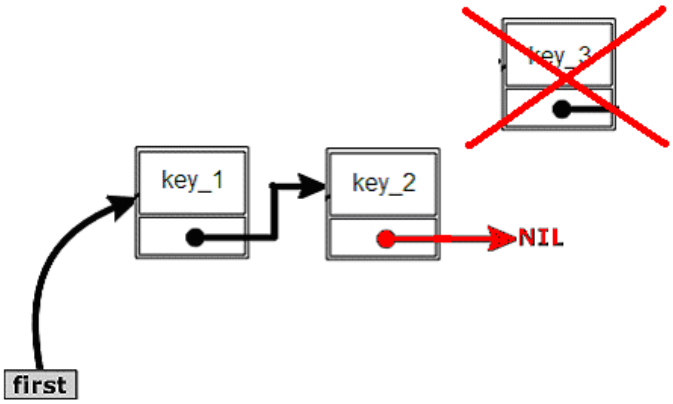
a) z początku



b) ze środka (usuwamy 2 element)



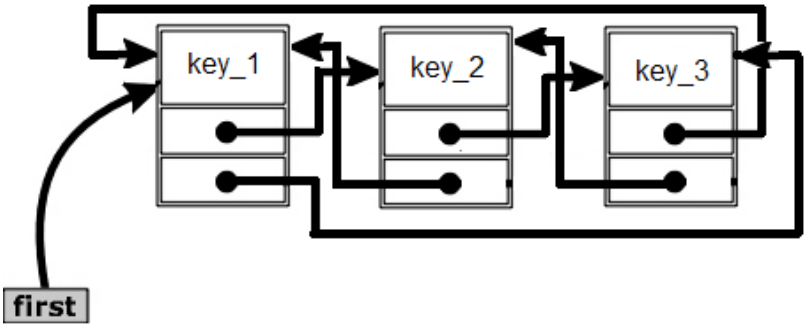
c) z końca



- *Dwukierunkowa*

Lista dwukierunkowa różni się od listy jednokierunkowej tylko tym, że każdy element w swojej strukturze oprócz klucza identyfikacyjnego posiada dwa wskaźniki *next* (wskazuje na następny element na liście) i *previous* (wskazuje na poprzedni element na liście) – dla porównania lista jednokierunkowa posiada tylko wskaźnik *next*.

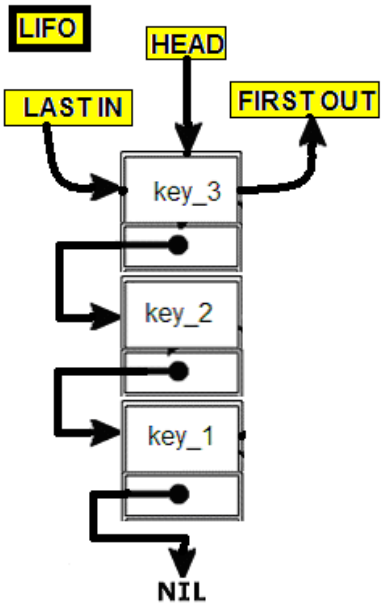
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.



Na liście dwukierunkowej można wykonywać operacje dodawania, usuwania elementów oraz przeszukiwania tej struktury w sposób analogiczny jak w przypadku listy jednokierunkowej. Trzeba tylko zwrócić baczną uwagę na fakt **zapewnienia spójności podczas dodawania lub usuwania elementów dla dwóch wskaźników**, a nie jak w przypadku listy jednokierunkowej dla jednego. Posiadanie dwóch wskaźników przez każdy element na liście powoduje, że brak ograniczenia związanego ze sposobem przeszukiwania listy (tylko od korzenia do liścia w przypadku listy jednokierunkowej), więc można się przesuwać po liście w obu kierunkach i z dowolnego miejsca, którego wskaźnik jest znany. Powyższy fakt powoduje, że struktura ta jest znacznie efektywniejsza pod względem przeszukiwania, ale niestety więcej czasu potrzebne jest na zarządzanie nowymi elementami. W formie ćwiczeń należy spróbować przerysować powyższe diagramy w taki sposób, aby odpowiadały wykonywanym operacjom dla listy dwukierunkowej.

STOS - LIFO (ANG. LAST IN FIRST OUT)

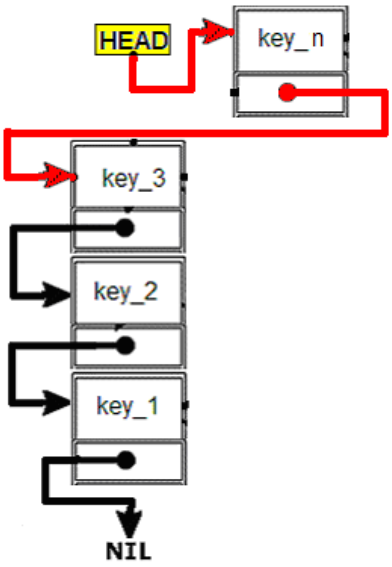
Stos jest szczególnym przypadkiem listy, w którym jedynie ostatni element, zwany *wierzchołkiem (ang. head)*, jest w danym momencie dostępny. Każde **dodanie** nowego elementu powoduje, że staje on się wierzchołkiem, który posiada wskaźnik na element, który był poprzednim wierzchołkiem. Jedynym elementem, który może być **usunięty** bez problemu w tej strukturze to wierzchołek. Jeżeli chce się usunąć element ze środka stosu to należy zdjąć wszystkie elementy (np.: na inny stos), które spowodują, że ten element, który chcemy usunąć stanie się wierzchołkiem, usunąć go a następnie przełożyć ponownie wszystkie elementy z tego tymczasowego stosu na ten właściwy.



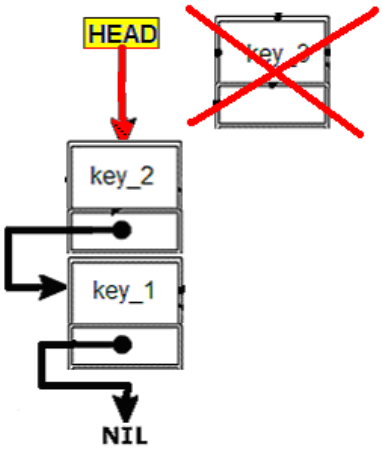
Podczas usuwania rekordu będącego wierzchołkiem należy zapewnić, że nowym wierzchołkiem zostanie rekord, na który usuwany wierzchołek wskazywał jako na następny. Stos jest bardzo często wykorzystywaną strukturą danych. Działanie na nim jest często porównywane do stosu kartek: nie można usunąć kartki znajdującej się na dnie stosu nie usuwając wcześniej wszystkich innych, gdyż cały stos nam się *„rozleci”*. Nie można także dodać nowej kartki gdzieś indziej, niż na samą górę. Stos można wykorzystać w algorytmie zmieniającym notację zapisu liczb z infiksowej na Odwrotną Notację Polską.

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

DODAWANIE NOWEGO ELEMENTU

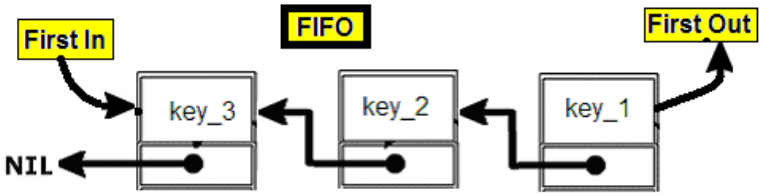


USUWANIE WIERZCHOŁKA STOSU



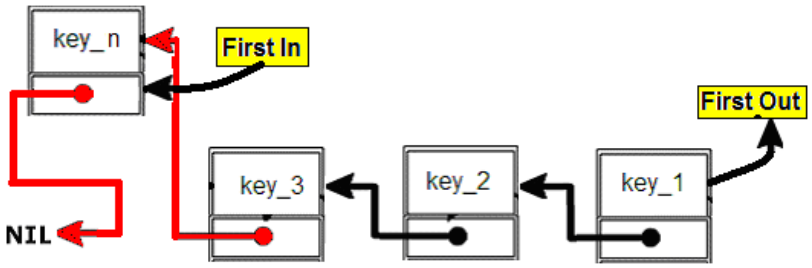
KOLEJKA - FIFO (ANG. FIRST IN FIRST OUT)

Kolejka jest szczególnym przypadkiem listy, w której nowe elementy można jedynie **dodawać na koniec kolejki**, a **usuwać można jedynie z jej początku**.



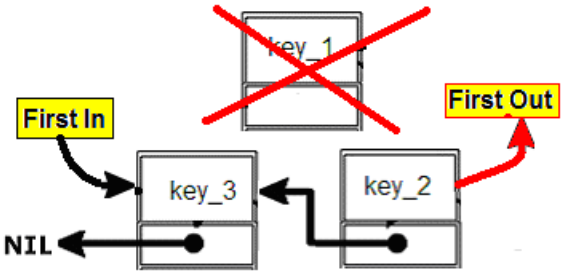
Procedura usunięcia danych z końca kolejki jest podobna, jak w przypadku **stosu**, z tą różnicą, że usuwane są dane od początku a nie od końca. Pierwszy element (a dokładniej wskaźnik do jego miejsca w pamięci) musi zostać zapamiętany, by możliwe było szybkie usuwanie pierwszego elementu. W przeciwnym razie, aby dotrzeć do pierwszego elementu należy przejść całą kolejkę od elementu aktualnego (czyli ostatniego). Możliwe działanie na kolejce są intuicyjnie jasne, gdyż można w prosty sposób skojarzyć ją z kolejką ludzi np.: w sklepie. Każdy nowy klient staje na jej końcu, obsługa odbywa się jedynie na początku.

DODAWANIE NOWEGO ELEMENTU



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

USUWANIE PIERWSZEGO ELEMENTU KOLEJKI



DRZEWO BST (**B**INARY **S**EARCH **T**REE)

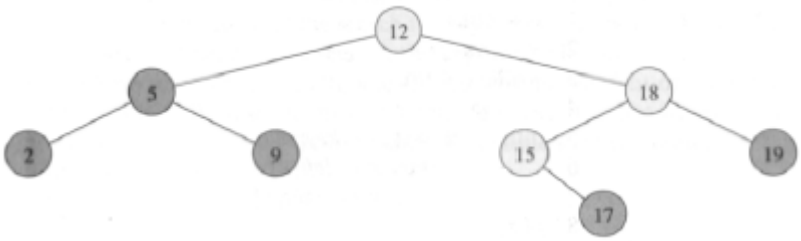
Jest drzewem binarnym, czyli takim, w którym każdy wierzchołek ma co najwyżej dwóch synów. Dodatkowo jest ono drzewem uporządkowanym, tzn. ważna jest kolejność jego synów (w przypadku drzew binarnych pierwszego syna nazywa się *lewym*, a drugiego *prawym*). Charakteryzuje się ono następującą własnością - lewe poddrzewo dowolnego wierzchołka zawiera wierzchołki o mniejszej wartości, a prawe zawiera wierzchołki o większej wartości od jego samego (w przypadku wartości równych wartości wierzchołka można wybrać, do którego drzewa mają należeć, ale trzeba następnie konsekwentnie postępować zgodnie z dokonanym wyborem).

TWORZENIE DRZEWA POPRZEZ DODAWANIE NOWYCH ELEMENTÓW

Stworzenie drzewa *BST* z podanej tablicy należy wykonywać poprzez wstawianie jej kolejnych elementów na to drzewo w taki sposób, aby za każdym spełnione były opisane powyżej zależności Realizuje się to przez wykonywanie następujących czynności dla wszystkich elementów tablicy:

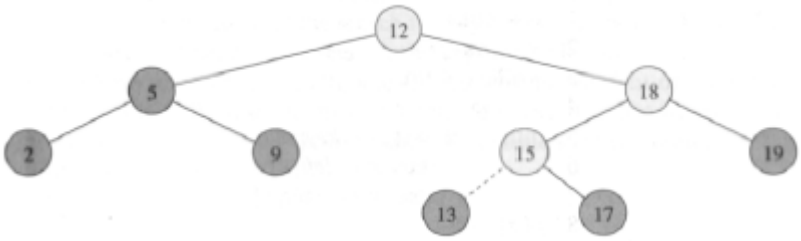
1. Jeśli drzewo *BST* jest puste to wstawiany jest element, który będzie pełnił rolę korzenia (i pomijane są następne punkty i pobierany jest kolejny element z tablicy). Gdy drzewo nie jest puste to porównujemy wstawiany element z korzeniem.
2. Jeżeli wartość elementu jest mniejsza od wartości porównywanego wierzchołka to następuje przejście w głąb drzewa do lewego syna, jeżeli zaś większa to do prawego syna (gdy równa to zależnie od tego jak jaka decyzja została podjęta, byle konsekwentnie).
3. Gdy tam gdzie *"doszliśmy"* nie ma żadnego wierzchołka to przechodzimy do następnego punktu. Jeżeli jednak znajduje się tam inny wierzchołek drzewa to musimy porównać go z wstawianym elementem i wrócić do punktu 2.
4. Skoro doszliśmy do *"wolnego miejsca"* to właśnie tam wstawiamy nasz element. Oczywiście będzie on synem wierzchołka, z którego przyszliśmy (a czy lewym czy prawym to zależy od tego, w którą stronę poszliśmy z tamtego wierzchołka).

Przykład: Załóżmy, że mamy częściowo stworzone drzewo w postaci zaprezentowanej na poniższym rysunku (np.: 12, 18, 5, 19, 2, 15, 9, 17):



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

Naszym kolejnym zadaniem jest dodanie wartości 13 do powyższego drzewa. W rezultacie uzyskamy drzewo zaprezentowane na poniższym rysunku.



WYSZUKIWANIE ELEMENTÓW

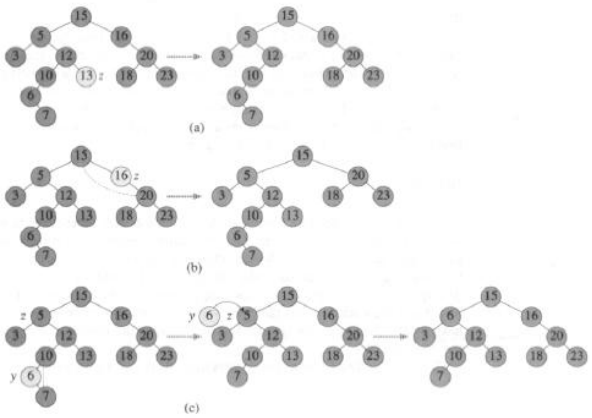
Po zbudowaniu drzewa wyszukanie wierzchołka o konkretnej wartości jest proste. Zaczynamy szukanie od korzenia. Porównujemy szukaną wartość z wartością wierzchołka, na którym się znajdujemy - jeśli jest mniejsza to idziemy głębiej w lewo, jeśli większa to głębiej w prawo. Czynność powtarzamy do czasu, gdy znajdziemy właściwą wartość lub, gdy dojdziemy do wierzchołka, którego nie możemy iść nigdzie głębiej (wtedy nie istnieje w drzewie szukana wartość).

USUWANIE ELEMENTÓW

Poza operacjami wstawiania i wyszukiwania elementu w drzewie *BST* istnieje także operacja usuwania z drzewa (potrzebna dużo rzadziej). Najpierw oczywiście trzeba odnaleźć element do usunięcia, a potem go usunąć. Nie muszę chyba dodawać, że należy to zrobić w taki sposób, aby ciągle były spełnione wszystkie powyżej opisane własności drzewa *BST*. Można wyróżnić trzy przypadki:

1. *Wierzchołek usuwany nie ma synów.*
   Wtedy po prostu go usuwamy.
2. *Wierzchołek usuwany ma jednego syna.*
   Po usunięciu wierzchołka należy jego jedynego syna (z całym jego poddrzewem) *"podczepić"* w miejsce usuwanego wierzchołka. Oznacza to po prostu, że syn usuwanego wierzchołka staje się synem ojca usuwanego wierzchołka.
3. *Wierzchołek usuwany ma dwóch synów.*
   W miejsce usuwanego wierzchołka należy *"podrzucić"* wierzchołek o najmniejszej wartości z prawego poddrzewa usuwanego (lub o największej wartości z lewego poddrzewa). Ten podrzucany wierzchołek jest najbardziej *"lewym"* (lub najbardziej *"prawym"*) w poddrzewie, co daje nam gwarancję, że ma co najwyżej jednego syna i można go usunąć zgodnie z punktem 1 lub 2.

Przykład: Usuwamy wierzchołek *z* drzewa *BST* w trzech możliwych przypadkach: (a) jeżeli wierzchołek *z* nie ma dzieci to go po prostu usuwamy, (b) jeżeli wierzchołek *z* ma tylko jedno dziecko to go usuwamy i podczepiamy jego dziecko w jego miejsce, (c) jeżeli wierzchołek *z* ma dwoje dzieci wtedy podłączamy w jego miejsce jeden z jego następników, który posiada co najwyżej jedno dziecko.



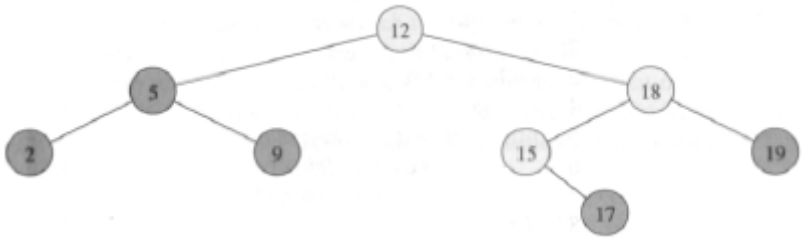
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

PRZESZUKIWANIE DRZEWA

Przeszukiwanie drzewa jest niczym innym jak podróżą po jego wierzchołkach w odpowiedniej kolejności. Czasem nazywa się to także numerowaniem drzewa, jako że najczęściej przydziela się wierzchołkom numery zgodnie z kolejnością przechodzenia.
Są trzy metody przeszukiwania drzewa:

1. **Pre-order** (*wzdłużna*): korzeń, lewe poddrzewo, prawe poddrzewo.
2. **In-order** (*poprzeczna*): lewe poddrzewo, korzeń, prawe poddrzewo.
3. **Post-order** (*wsteczne*): lewe poddrzewo, prawe poddrzewo, korzeń.

Dla wyjaśnienia skoncentrujmy się na pierwszej metodzie. Najpierw przydzielamy numer korzeniowi naszego drzewa, później przydzielamy kolejne numery całemu lewemu poddrzewu (tzn. wszystkim jego wierzchołkom) i dopiero później całemu prawemu poddrzewu dalsze numery. Przydzielanie numerów poddrzewu wykonuje się tą samą metodą: najpierw korzeniowi (dodam, że korzeń poddrzewa jest synem korzenia naszego drzewa), później jego lewemu poddrzewu i wtedy prawemu. Najlepiej się to wykonuje rekurencyjnie.
Jeśli chodzi o zastosowanie to najprostsze ma metoda druga. Otóż wypisując wierzchołki w kolejności przechodzenia metodą **In-Order** otrzymamy ciąg posortowany. Pozostałe metody oczywiście mają też zastosowanie, ale nieco bardziej zaawansowane (zazwyczaj przydają się do różnych ciekawych algorytmów na grafach).
Przykład: Załóżmy, że mamy stworzone drzewo w postaci zaprezentowanej na poniższym rysunku



Pre-order: 12, 5, 2, 9, 18, 15, 17, 19
In-order: 2, 5, 9, 12, 15, 17, 18, 19
Post-order: 2, 9, 5, 17, 15, 19, 18, 12

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:

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

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:

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.

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

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.

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

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

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.

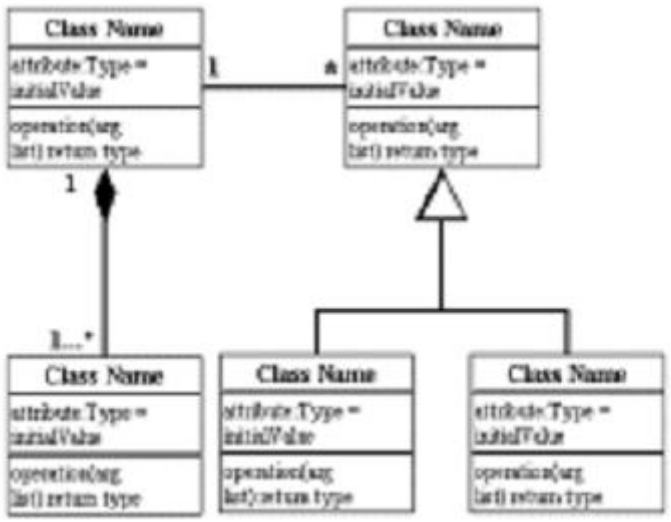
Diagramy te są podzielone na cztery rodzaje diagramów abstrakcyjnych:

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

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

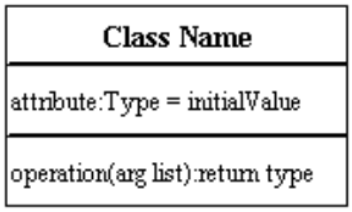
Jest podstawą prawie każdej metody zorientowanej obiektowo. Opisuje statyczną budowę systemu.



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

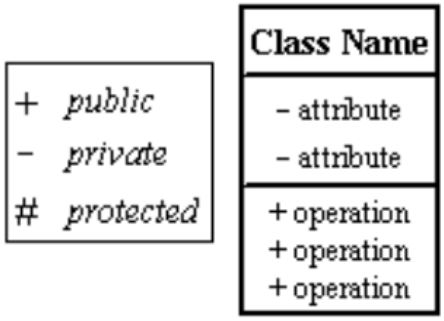
Klasy – przedstawione jako prostokąty podzielone na trzy części; zawierają nazwę klasy w pierwszym pod2prostokącie, listę atrybutów w drugim oraz operacje w trzecim.

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

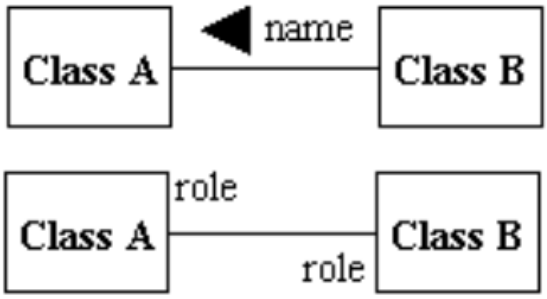


Widoczność – określa poziom dostępu do informacji zawartej w danej klasie;

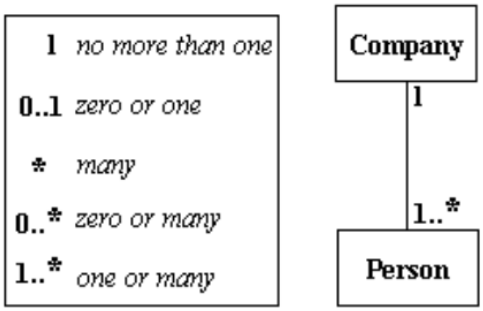
- *private* ukrywa informacje przed innymi klasami z zewnątrz,
- *public* pozwala innym klasom widzieć informacje danej klasy,
- *protected* pozwala dzieciom (klasom dziedziczącym) danej klasy na dostęp do informacji, a ukrywa je przed klasami zewnętrznymi.



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.

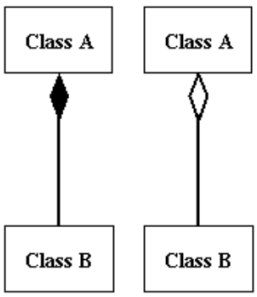


Liczność – umieszczane są na końcach asocjacji; określają liczbę instancji klasy połączonej z instancją innej klasy.



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

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.

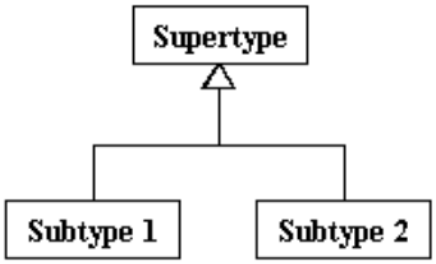


DIAGRAM OBIEKTÓW

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

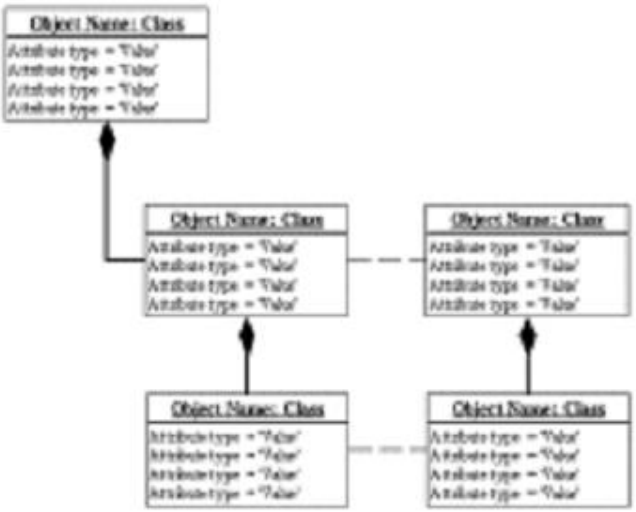


DIAGRAM PAKIETÓW

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.

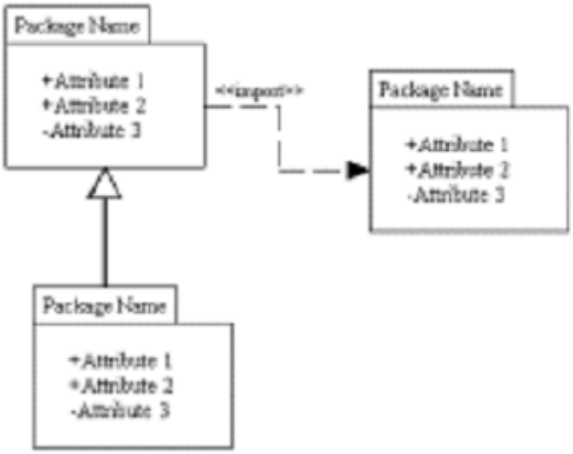
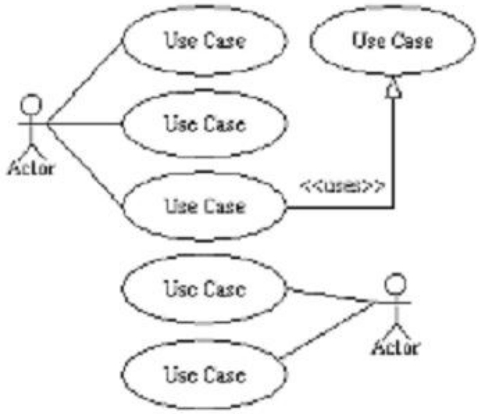


DIAGRAM PRZYPADKÓW UŻYCIA

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:

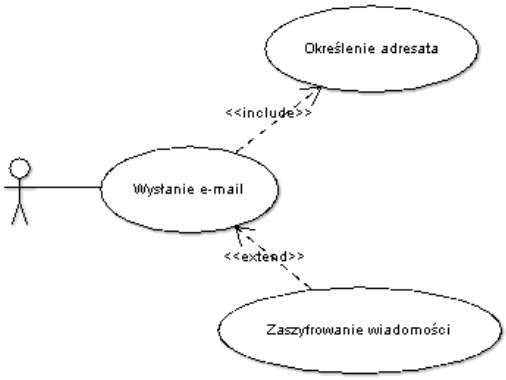


DIAGRAM STRUKTUR ZŁOŻONYCH

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

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

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.

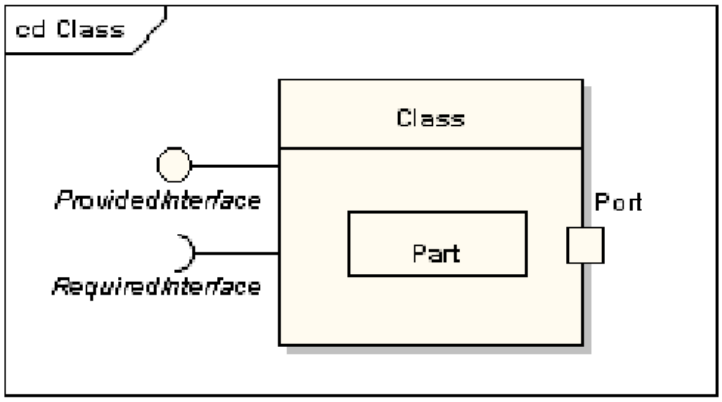


DIAGRAM KOMPONENTÓW

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

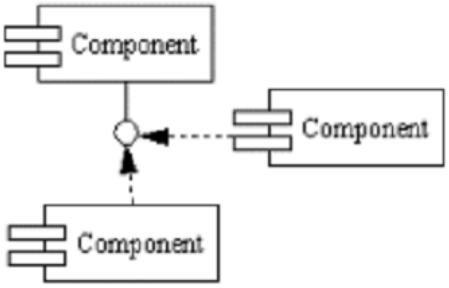
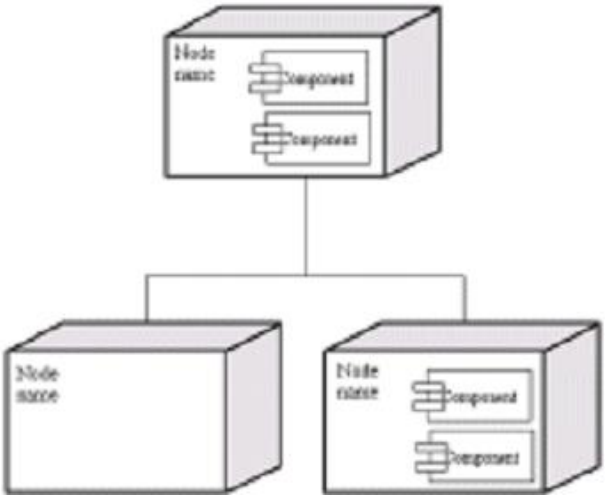


DIAGRAM WDROŻENIOWY

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



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

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.

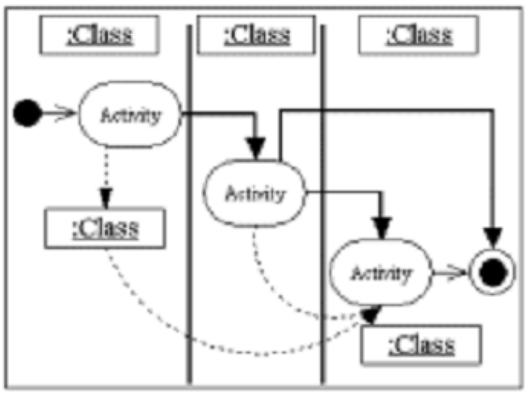


DIAGRAM STANÓW (MASZYNY STANOWEJ)

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

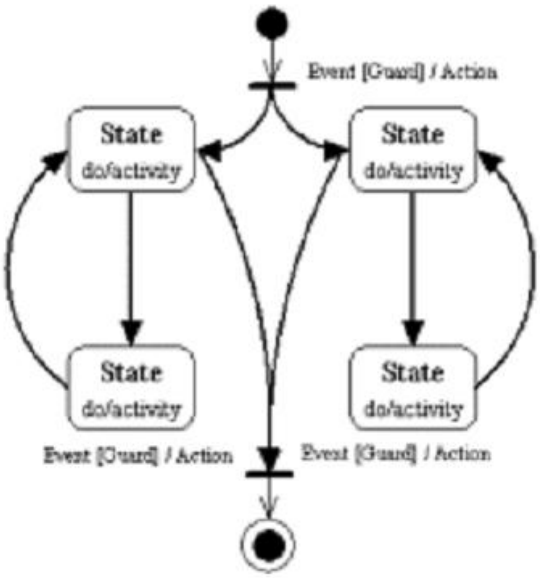


DIAGRAM ZALEŻNOŚCI CZASOWYCH

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.

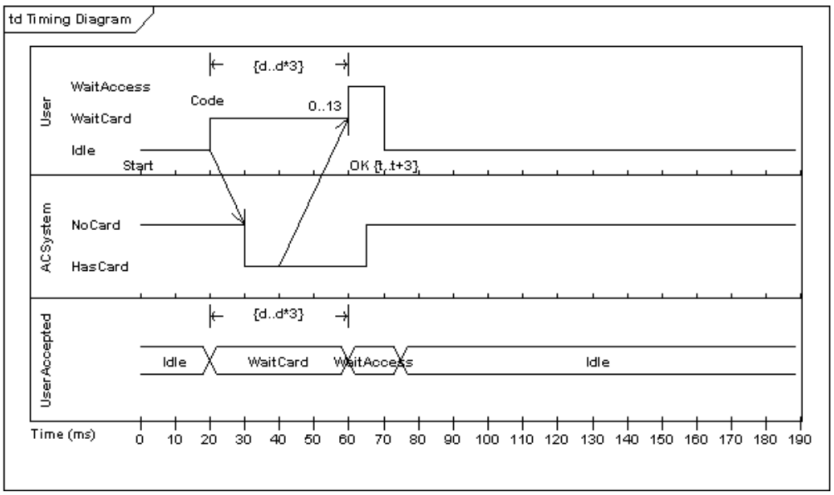
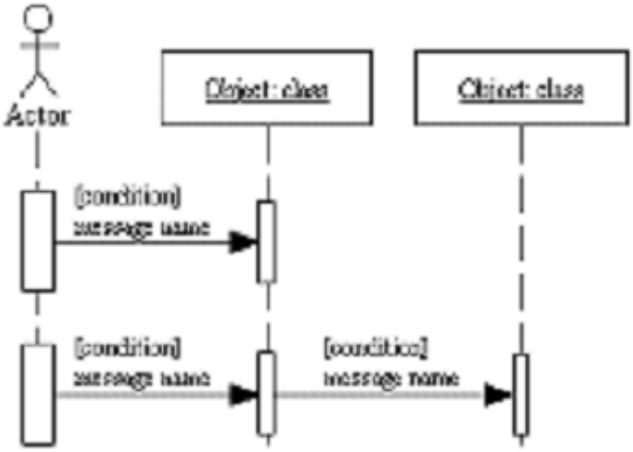


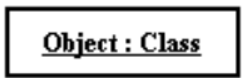
DIAGRAM SEKWENCJI

Opisują interakcje pomiędzy klasami w znaczeniu wymiany wiadomości w czasie.

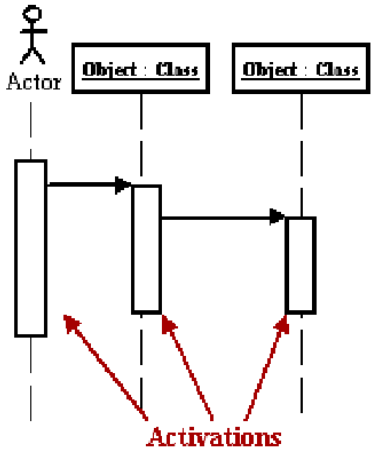


Poniżej przedstawione zostały podstawowe elementy, które składają się na diagram sekwencji.

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

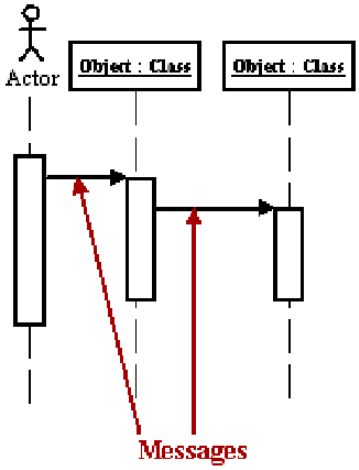


Aktywacja – prostokąty reprezentujące czas, jaki obiekt potrzebuje do zakończenia zadania.



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

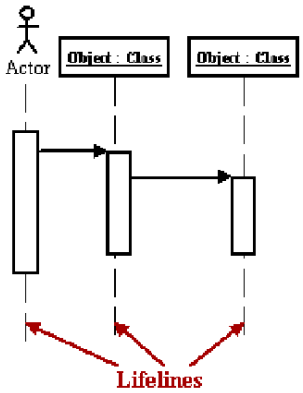
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.



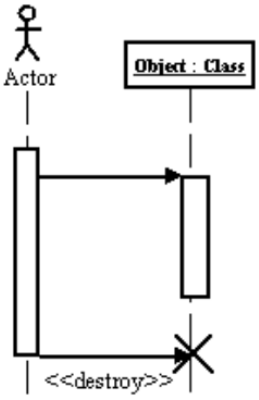
Wiadomości asynchroniczne są wysyłane z obiektu, który nie czeka na odpowiedź od odbiorcy wiadomości przed kontynuowaniem swoich zadań.

Arrow	Message type
→	Simple
⟶	Synchronous
⟶	Asynchronous
↩	Balking
⟶	Time out

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

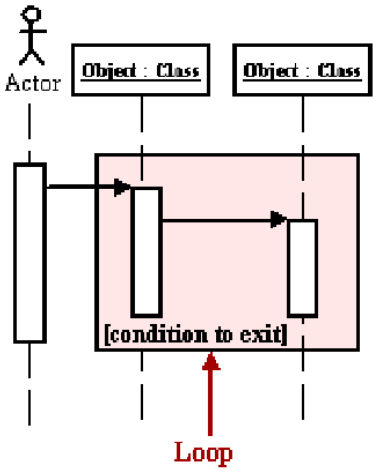


Zniszczenie obiektów – obiekty mogą być zakończone wcześniej poprzez użycie strzałki oznaczonej przez *„<<destroy>>”*, która wskazuje na X.



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

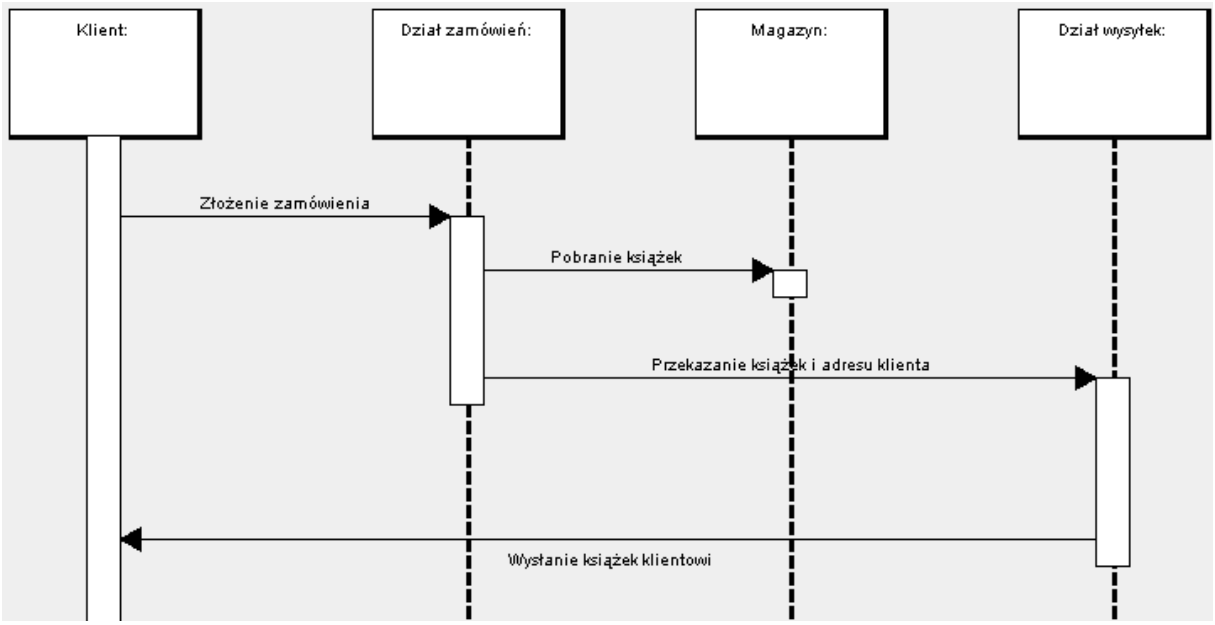
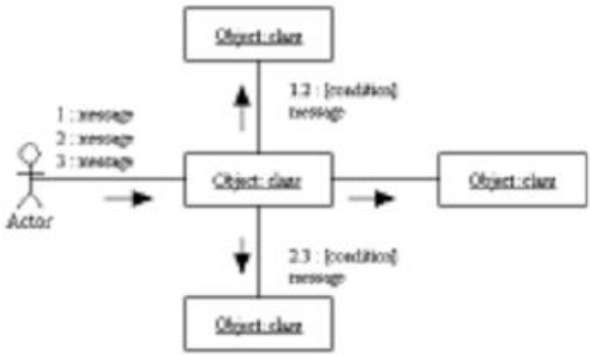


DIAGRAM KOMUNIKACJI (KOLABORACJI)

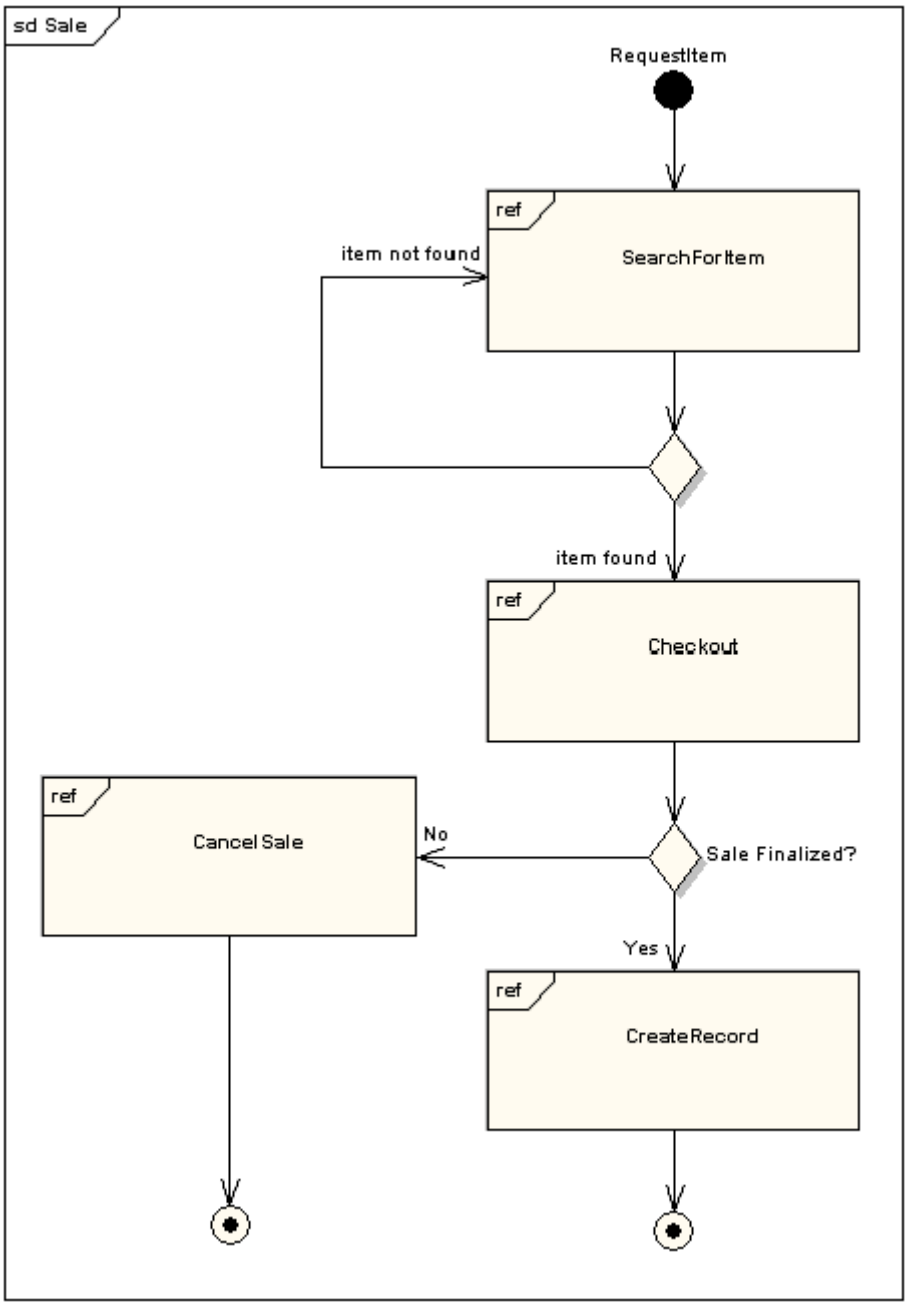
Reprezentuje interakcje pomiędzy obiektami jako serie kolejnych wiadomości. Opisują zarówno statyczną strukturę jak i dynamiczne zachowanie systemu.



(*) 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.



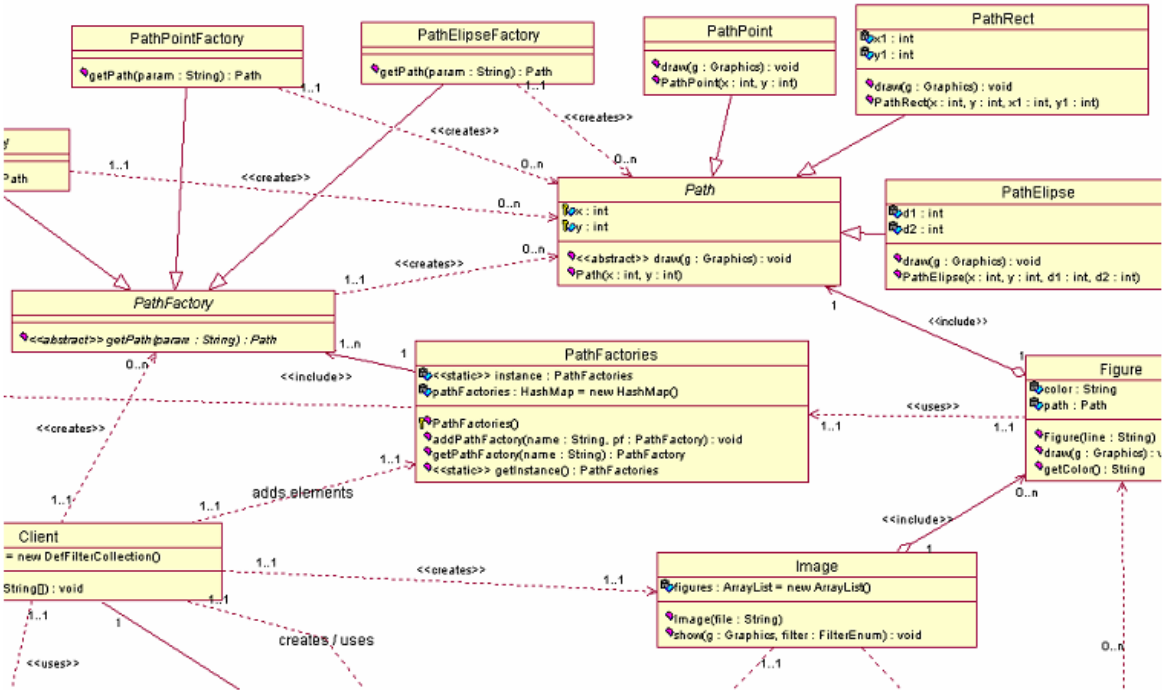
PRZYKŁADOWY PROJEKT I JEGO ROZWIĄZANIE

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

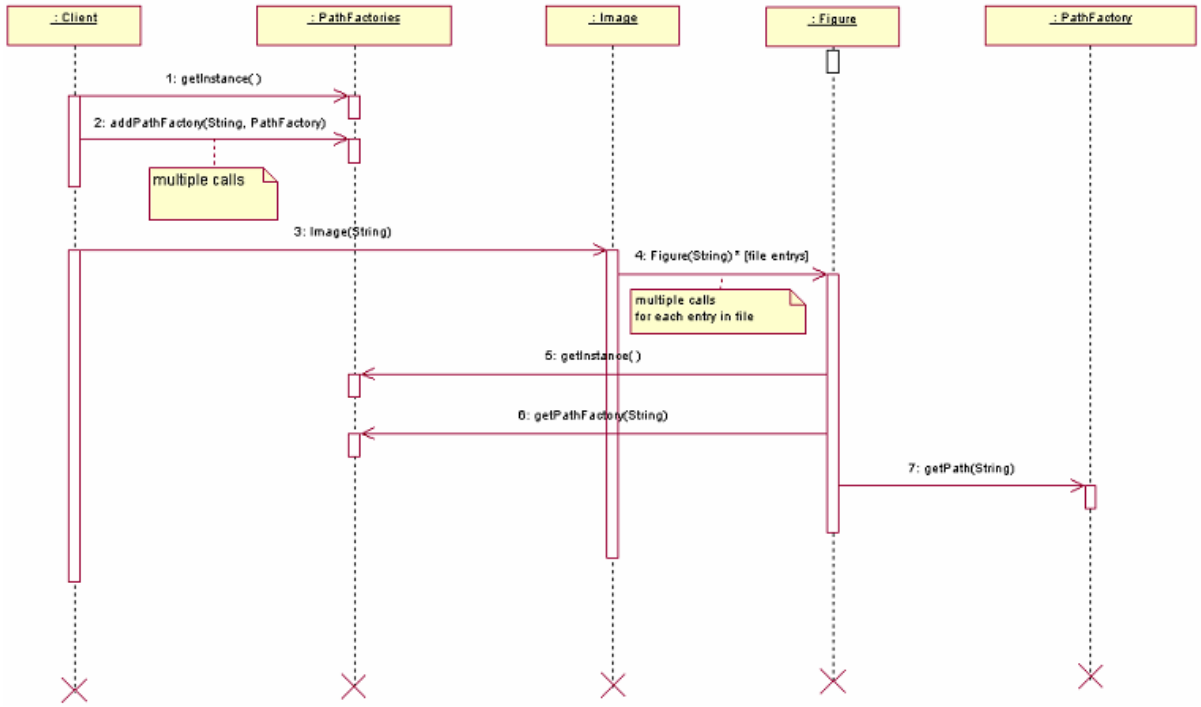
(*) gwiazdką oznaczone są zadania, które nie są realizowane na ćwiczeniach i są przeznaczone do wykonania jako zadania domowe.

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.