Zajęcia 2. Dodanie do szablonu BibliotekaKlas klasy Lista

2.1. Wstęp

Poprawne tworzenie i inicjalizowanie obiektów typu `Pracownik` jest warunkiem koniecznym do zapewnienia pełnej funkcjonalności tworzonej aplikacji (zarówno w wersji pod konsolę jak i okienkowej). Kolejnym etapem budowy programu jest umożliwienie dokonywania operacji na pewnej liczbie pracowników ujętych w listę. Do takich operacji należeć powinny: dodawanie, usuwanie, wyszukiwanie pracowników jak również sortowanie, wczytywanie z klawiatury i wypisywanie na ekran ich danych oraz wykonywanie zapisu oraz odczytu informacji z pliku. Aby to umożliwić wygodnie jest stworzyć klasę `Lista`, która udostępni odpowiednie metody służące do wykonywania wyżej wymienionych czynności na liście. Klasę tę należy dodać do projektu **BibliotekaKlas** w analogiczny sposób jak klasy zdefiniowane w poprzednim rozdziale.

2.2. Dodanie klasy `Lista` do biblioteki klas

Aby zapewnić bezpieczeństwo typów i wyeliminować ewentualną konieczność rzutowania lub sprawdzania typów obiektów, klasa `Lista` musi posiadać pole składowe, które przechowuje elementy w formie kolekcji generycznej. Kolekcje generyczne zawarte są w przestrzeni `System.Collections.Generic` i udostępniają zbliżoną funkcjonalność co zwykłe kolekcje (przestrzeń `System.Collections`) – czyli kontenery przechowujące obiekty typu `Object` (np.: klasa `ArrayList`). Jak wspomniano we wstępie, tworzona aplikacja ma umożliwiać wykonywanie operacji na grupie pracowników (tj. na obiektach typu `Pracownik`), dlatego typ generyczny kolekcji należy zdefiniować jako `<Pracownik>`. W związku z tym w klasie `Lista` musi zostać zadeklarowane (prywatne) pole składowe typu `List<T>`, gdzie `T` oznacza typ `Pracownik`. Będzie ono służyć jako kontener do przechowywania obiektów typu `Pracownik` (a w przyszłości także typów pochodnych). Nazwa pola może być dowolna – w niniejszej książce przyjęto nazwę `lista`. Aby z kolei dokonywać wybranych operacji na liście, w klasie należy dodatkowo zdefiniować poniższe publiczne metody:

- Konstruktor domyślny inicjalizujący pole składowe `lista` poprzez wywołanie konstruktora klasy `List`.
- `void Dodaj(Pracownik pracownik)` dodającą pracownika do listy pracowników.
- `void WstawWPolozenie(int indeks, Pracownik pracownik)` dodającą do listy pracownika przekazanego w argumencie metody w położenie `indeks`.
- `int Usun(string nazwisko)` usuwającą z listy pracownika o zadanym w argumencie nazwisku i zwracającą indeks usuniętego pracownika. Jeżeli na liście nie znajduje się pracownik o danym nazwisku, metoda ma zwrócić `-1`.
- `void Usun(int indeks)` usuwającą pracownika z listy znajdującego się na pozycji `indeks`. W przypadku niepoprawnej wartości argumentu metody (np. wartość mniejsza od `0`), na ekranie pojawić się ma stosowny komunikat.
- `Pracownik Szukaj(string nazwisko)` wyszukującą na liście pracownika o zadanym w argumencie nazwisku. Jeżeli na liście znajduje się taki pracownik, metoda ma zwrócić znaleziony obiekt. W przeciwnym wypadku metoda ma zwrócić `null`.
- `void Sortuj()` sortującą elementy listy.
- `void ZapisConsole()` wypisującą na ekran wszystkich pracowników z listy.
- `void OdczytConsole()` wczytującą z klawiatury dane do nowostworzonego obiektu klasy `Pracownik` i dodającą obiekt do listy.
- `void Wyczysc()` czyszczącą listę, tj.: usuwającą z listy wszystkie obiekty typu `Pracownik`.

W klasie należy również zdefiniować dwie właściwości:

- `public Pracownik this[int i]` – indekser zwracający `i`-ty element listy.
- `public int Rozmiar` zwracającą liczbę elementów listy.

2.3. Klasa `List<T>` i jej wybrane metody

Jak już wspomniano w podrozdziale 2.2, klasa `List<T>` jest generycznym odpowiednikiem klasy `ArrayList`, czyli kolekcji typu tablica. Przez twórców platformy .NET została zaprojektowana tak, iż jej rozmiar jest automatycznie zwiększany w miarę potrzeb. Informację na temat ilości przechowywanych elementów w kolekcji można uzyskać poprzez odwołanie się do właściwości `Capacity`. Podczas inicjalizacji obiektu typu `List<T>`, właściwość ta przyjmuje wartość `0`, a po dodaniu pojedynczego elementu zwiększana jest do `4`. Po każdym zapełnieniu kolekcji zwiększana jest o `4`.
Klasa `List<T>` definiuje ponadto szereg właściwości i metod, które mogą zostać wykorzystywane do przeszukiwania i modyfikowania zawartości kolekcji. Operacje na tablicy obejmują takie działania jak wstawianie i usuwanie w wybranych miejscach, dodawanie, czyszczenie, poszukiwanie określonych obiektów na liście. W przeciwieństwie do kolekcji typu `ArrayList`, klasa udostępnia również mechanizm obsługi wyrażeń lambda (=>). Dzięki temu w bardzo prosty sposób można na przykład zaimplementować operacje przeszukiwania tablicy. Poniżej przedstawiono kilka wybranych składowych kolekcji `List<T>`, które warto wykorzystać podczas definiowania metod oraz właściwości klasy `Lista`:

- `void Add(T t)`: metoda służąca do dodawania obiektu `t` na koniec listy (dla typów referencyjnych `t` może przyjmować wartość `null`).
- `void Insert(int indeks, T t)`: metoda służąca do wstawiania obiektu `t` we wskazane przez argument `indeks` miejsce (numeracja elementów kolekcji rozpoczyna się od indeksu `0`).
- `bool Remove(T t)`: metoda służąca do usuwania obiektu `t` z listy. Warto zwrócić uwagę, że metoda zwraca wartość typu `bool`: `true` w przypadku, kiedy obiekt `t` został poprawnie usunięty z listy, `false` wówczas, gdy obiekt nie został znaleziony w kolekcji.
- `void RemoveAt(int indeks)`: metoda służąca do usuwania obiektu ze wskazanego przez argument `indeks` miejsca.
- `void Sort()`: metoda służąca do sortowania elementów listy. Za pomocą właściwości domyślnego komparatora `Comparer<T>.Default` dla typu `T` ustala kolejność elementów. W tym celu sprawdzane jest, czy typ `T` implementuje interfejs `IComparable<T>`. Jeżeli tak, do celów sortowania używana jest implementacja interfejsu. W przeciwnym wypadku właściwość `Comparer<T>.Default` sprawdza czy typ `T` implementuje interfejs `IComparable`. Jeżeli kompilator wykryje brak odpowiedniej implementacji, rzucany jest wyjątek `InvalidOperationException`.
- `void Clear()`: metoda służąca do usuwania wszystkich elementów listy.
- `bool Contains(T t)`: metoda służąca do sprawdzenia czy w kolekcji znajduje się obiekt `t` przekazany w argumencie. Jeżeli taki obiekt istnieje – zwracana jest wartość `true`, w przeciwnym wypadku – `false`.

Wskazówka: Aby zdefiniować wybrane metody dla klasy `Lista`, warto skorzystać z wyżej opisanych metod zaimplementowanych w klasie `List<T>`. Można je wywoływać na rzecz obiektu tej klasy.

2.4. Definicja wybranych metod klasy `Lista`

Dodawanie pracownika do listy

Argumentem metody `Dodaj`, którą należy zdefiniować dla klasy `Lista` jest obiekt typu `Pracownik`. Na pierwszy rzut oka wydawałoby się zatem, że wywołanie w tej metodzie metody `Add` (opisanej w podrozdziale 2.3) klasy `List<T>` i przesyłanie jej jako argument tego właśnie obiektu wystarczy do dodania elementu do listy. Takie rozwiązanie jest poprawne wyłącznie w przypadku, gdy metoda `Dodaj` wywołana jest w sposób przedstawiony na Listingu 2.1.

```
Lista l = new Lista();
l.Dodaj(new Pracownik("Jan","Kowalski",1,"lutego" 1999,"Mickiewicza","1","Krakow"));
l.Dodaj(new Pracownik("Piotr","Nowak",2,"maja",2001,"Sienkiewicza","2","Rzeszow"));
```

Listing 2.1: Dodawanie do listy obiektów tworzonych przy użyciu konstruktora klasy `Pracownik`.

Jej argumentem jest wówczas instancja nowego obiektu stworzonego za pomocą operatora `new`. Dodawanie pracownika do listy nie musi jednak polegać na wielokrotnym wywołaniu konstruktora klasy `Pracownik` przy każdym wywołaniu metody `Dodaj`. Do listy obiekty można przecież dodawać w prostszy sposób (bez konieczności tworzenia nowego egzemplarza klasy za każdym razem), pozwalając dodatkowo użytkownikowi wpisywać dowolne dane z klawiatury. Kod źródłowy przedstawiony na Listingu 2.2 obrazuje bardziej elastyczne rozwiązanie.

```
Pracownik p = new Pracownik();
p.OdczytConsole();
l.Dodaj(p);
p.OdczytConsole();
l.Dodaj(p);
```

Listing 2.2: Wielokrotne dodawanie do listy jednego obiektu typu `Pracownik` o zmodyfikowanych składowych.

W tym wypadku obiekt klasy `Pracownik` tworzony jest wyłącznie raz (wywołanie konstruktora domyślnego) i po wprowadzeniu odpowiedniej informacji z klawiatury poprzez wywołanie metody `OdczytConsole`, dodawany jest do listy. Poprawne działanie powyższego kodu (mimo, że nie ma tutaj błędów zarówno składniowych jak i logicznych) jest uzależnione od definicji metody `Dodaj` klasy `Lista`. Jeżeli metoda zdefiniowana jest w sposób pokazany na Listingu 2.3,

```
public void Dodaj(Pracownik pracownik)
{
   lista.Add(pracownik);
}
```

Listing 2.3: Niepoprawna definicja metody `Dodaj`.

to przy każdym wywołaniu metody `OdczytConsole` na rzecz obiektu `p`, dojdzie do modyfikacji zawartości całej kolekcji: wszystkie elementy przyjmą wartości ostatnio modyfikowanego obiektu `p`. Stanie się tak dlatego, że metoda `Add` pracuje na referencji dodawanego obiektu, czyli na jego oryginale. Wpisanie nowych danych dla pracownika w programie i dodanie go do listy pociąga za sobą modyfikację całej grupy pracowników – jest to zjawisko bardzo niebezpieczne, gdyż nie jest kontrolowane przez użytkownika. Zatem jak zdefiniować metodę `Dodaj`, aby kolejne wywołania metody `OdczytConsole` nie niszczyły zgromadzonych do tej pory danych?

Odpowiedź na to pytanie można znaleźć w różnicy wywołań metody `Add` na rzecz obiektu `lista` oraz metody `Dodaj` na rzecz obiektu `l`. Warto zwrócić uwagę na to, że jeżeli argumentem przesyłanym do metody `Dodaj` jest obiekt klasy `Pracownik` tworzony za pomocą operatora `new`, to wówczas metoda `Add` dodaje nowo stworzoną instancję do kolekcji typu `List<Pracownik>` i nie dochodzi do jakiejkolwiek modyfikacji elementów listy. Natomiast w przypadku przesyłania raz stworzonego obiektu, który ulega zmianom poprzez interfejs z użytkownikiem, metoda `Add` dodaje ten sam obiekt a elementy kolekcji stają się identyczne. Rozwiązanie tego problemu tkwi w argumencie przesyłanym do metody `Add`. Metodę tę należy wywołać w taki sposób, by po każdym wywołaniu metody `Dodaj` po modyfikacji obiektu `p`, do listy dodawać nowy obiekt, a nie ten sam. Ale skąd przy każdym wywołaniu metody `Dodaj` brać nowy obiekt typu `Pracownik`? Najprostszym rozwiązaniem jest przesłanie do metody `Add` jako argument instancji typu `Pracownik` stworzonej na wzór argumentu metody `Dodaj`. Będzie to nic innego jak wywołanie konstruktora kopiującego tej klasy (rozdział 1.6). Metoda `Dodaj` przyjmie postać taką jak na Listingu 1.5.

```
public void Dodaj(Pracownik pracownik)
{
   lista.Add(new Pracownik(pracownik));
}
```

Listing 2.4: Poprawna definicja metody `Dodaj`.

Rozwiązanie zilustrowane na Listingu 2.4 zapewni poprawne dodawanie pracowników do listy bez jakichkolwiek niebezpiecznych (i niekontrolowanych) modyfikacji jej elementów i na tym etapie działania i funkcjonalności aplikacji jest jak najbardziej wystarczające. Teraz metodę `Dodaj` można wywoływać na obydwa sposoby: przesyłając jej tworzony obiekt (wywołania konstruktora) lub obiekt już istniejący, do którego użytkownik wielokrotnie podaje jakieś dane.

Należy jednak już w tym momencie podkreślić, że zaprezentowane podejście nie jest uniwersalne z punktu widzenia idei programowania obiektowego. Ze względu na to, że klasa `Pracownik` będzie stanowić typ podstawowy dla nowych, bardziej wyspecjalizowanych klas, metodę `Dodaj` dobrze jest już teraz ulepszyć do postaci zaprezentowanej na Listingu 2.5.

```
public void Dodaj(Pracownik pracownik)
{
   lista.Add(pracownik.Clone());
}
```

Listing 2.5: Optymalna definicja metody `Dodaj` umożliwiająca zastosowanie polimorfizmu przy dodawaniu obiektów do listy.

Jak można zauważyć różnica w definicji metody `Dodaj` sprowadza się jedynie do tego, że zamiast wywołania konstruktora kopiującego klasy `Pracownik`, do metdody `Add` przysyłany jest rezultat zwracany przez metodę `Clone` wywołaną na rzecz obiektu `pracownik`. Metoda `Clone` jest publiczną wirtualną metodą składową klasy `Pracownik`, która ma zwracać instancję nowostworzonego obiektu na wzór referencji `this` (rozdział 1.6). Jest więc ona odpowiednikiem konstruktora kopiującego. Na chwilę obecną wprowadzona modyfikacja daje ten sam rezultat, ale na przyszłość – wprowadza polimorfizm i upraszcza strukturę programu.

Wskazówka: W podobny sposób należy zdefiniować metodę `WstawWPolozenie`, która dodaje do listy pracownika w wybrane miejsce.

Wyszukiwanie pracownika na liście

Znalezienie na liście pracownika o zadanym nazwisku sprowadza się do przeszukania zawartości całej kolekcji i sprawdzenia czy nazwisko kolejnego obiektu równe jest nazwisku przekazanemu do metody jako argument. Można w tym celu użyć pętli `foreach` lub `for` (`while`, `do while`). W momencie gdy pracownik o zadanym nazwisku zostanie znaleziony, metoda ma zwrócić jego instancję. W przypadku przejścia przez całą listę i nie znalezienia obiektu, wartością zwracaną metody ma być referencja `null`. Listing 2.6 przedstawia definicję metody `Szukaj`.

```
public Pracownik Szukaj(string nazwisko)
{
   foreach (Pracownik p in lista)
   {
      if (p.Nazwisko.Equals(nazwisko))
      {
         return p;
      }
   }
   return null;
}
```

Listing 2.6: Definicja metody `Szukaj` do znajdywania pracownika na liście o zadanym nazwisku.

Ze względu na to, że obiekt `lista` jest kolekcją generyczną typu `List<Pracownik>`, powyższy sposób wyszukiwania pracownika po nazwisku można uprościć omijając użycie pętli. Klasa `List<T>` (w porównaniu

z nie-generyczną kolekcją `ArrayList`) ma dużą ilość predefiniowanych metod, które ułatwiają dokonywanie różnych operacji na liście. Jedną z nich jest metoda `Find`. Metoda ta szuka elementu, który odpowiada warunkowi wyspecyfikowanemu za pomocą predykata (wyrażenia) i zwraca pierwsze wystąpienie obiektu w kolekcji. Najważniejsze jest tutaj poprawne zdefiniowanie predykata. Tak w rzeczywistości jest to generyczny delegat typu `System.Predicate<T>`, który jest używany jako osłona (opakowanie) każdej metody zwracającej wartość logiczną i przyjmującą wartość `T` jako jedyny parametr wejściowy. Metodę `Szukaj` można wówczas zdefiniować w sposób zaprezentowany na Listingu 2.7.

```
public Pracownik Szukaj(string nazwisko)
{
   Pracownik szukany = lista.Find(
   delegate(Pracownik p)
   {
      return (nazwisko.Equals(p.Nazwisko)) ? true : false;
   });
   return szukany;
}
```

Listing 2.7: Definicja metody `Szukaj` przy użyciu metody anonimowej.

W kodzie Listingu 2.7, argumentem metody `Find` jest metoda anonimowa (bez nazwy) zdefiniowana za pomocą słowa kluczowego `delegate`, której argumentem jest typ `Pracownik` a wartością zwracaną `bool`. Są to zatem typy zgodne z wymaganiami predykata. W przypadku, gdy porównywane nazwiska są identyczne, metoda anonimowa zwraca wartość `true`, a metoda `Find` podaje pierwsze wystąpienie obiektu na liście. W przeciwnym wypadku zwracana jest referencja zerowa `null`. Niestety zaprezentowana składnia metody anonimowej jest dość nieprzejrzysta, zawiła i wręcz odstraszająca. Powyższą składnię można na szczęście zupełnie uprościć poprzez zastosowanie wyrażeń lambda.

Wyrażenia lambda są niczym innym jak metodami anonimowymi. Można je przez to używać wszędzie tam, gdzie występują metody anonimowe lub delegaty o ściśle określonym typie. Wyrażenia lambda mogą zawierać zarówno same wyrażenia jak i deklaracje. Aby zdefiniować wyrażenie lambda, najpierw należy podać listę parametrów, następnie operator `=>` a na koniec zbiór jednej lub więcej instrukcji. Operator `=>` służy do odseparowania zmiennych wejściowych po swojej lewej stronie od ciała operatora po prawej stronie.

Na Listingu 2.8 zobrazowano użycie wyrażenia lambda do znalezienia w tablicy łańcuchów pierwszego wystąpienia łańcucha zawierającego literę łańcuch `"m"`.

```
string[] lancuchy = { "Ala", "ma", "kota" };
string literaM = lancuchy.First(x => x.Contains("m"));
```

Listing 2.8: Wyszukiwanie pierwszego wystąpienia łańcucha zawierającego podłańcuch `"m"` przy użyciu wyrażenia lambda.

Argument metody `First` można zinterpretować w następujący sposób: szukaj takiego łańcucha `x`, żeby `x` zawierał literę `m`. Należy podkreślić, że argumentem metody `First` jest wartość zwracana przez metodę `Contains`, czyli `bool`. Kod Listingu 2.9 prezentuje przykład, jak w tablicy liczb całkowitych można szybko znaleźć wszystkie liczby nieparzyste.

```
int[] liczby = { 0, 4, 1, 3, 9, 8, 5, 7, 2, 0 };
var liczbyNieParzyste = liczby.Where(x => x % 2 == 1);
```

Listing 2.9: Wyszukiwanie wystąpienia wszystkich liczb nieparzystych za pomocą wyrażenia lambda.

W tym wypadku metoda `Where` wyszukuje takie `x` (liczba typu `int`) dla którego wyrażenie (`x % 2 == 1`) stanowi prawdę.

Jak można zastosować wyrażenie lambda do metody wyszukującej pracownika na liście? Wystarczy tylko w metodzie `Szukaj` umieścić instrukcję przedstawioną na Listingu 2.10.

```
return lista.Find(x => x.Nazwisko.Equals(nazwisko));
```

Listing 2.10: Wyszukiwanie w liście obiektu, którego właściwość `Nazwisko` równe jest łańcuchowi `nazwisko`.

Tutaj, parametr `x` jest obiektem typu `Pracownik`. Wyszukiwany jest zatem w liście taki pracownik (`x`), którego nazwisko (`x.Nazwisko`) równe jest zadanemu w argumencie metody nazwisku. Jeżeli obiekt znajduje się na liście, metoda `Equals` zwróci `true` a metoda `Find` instancję znalezionego obiektu.

Sortowanie pracowników przy użyciu metody `Sort()`

Jak wspomniano w podrozdziale 2.3, aby móc wykorzystać metodę `Sort()` klasy `List<T>`, w metodzie `Sortuj()` klasy `Lista`, należy wcześniej zaimplementować interfejs `IComparable<T>` dla typu `T`. Ze względu na to, iż tworzony program powinien oferować możliwość sortowania pracowników po wybranych jego cechach (które zdefiniowane są poprzez odpowiednie pola, np.: `nazwisko`, `rok`, `ulica`), w tym wypadku `T` określa typ `Pracownik`. W związku z tym klasa `Pracownik` musi implementować interfejs `IComparable<Pracownik>`. Omawiana implementacja sprowadza się jedynie do poinformowania klasy `Pracownik`, że implementuje ona interfejs (poprzez umieszczenie nazwy interfejsu na liście dziedziczenia klasy) oraz do zdefiniowania jednej metody tego interfejsu, tj.:

```
int CompareTo(Pracownik other)
```

Przykładowo, jeżeli istnieje potrzeba posortowania listy pracowników po ulicy, w metodzie `CompareTo` w klasie `Pracownik` należy wpisać kod zaprezentowany na Listingu 2.11.

```
public int CompareTo(Pracownik other)
{
   return this.AdresZamieszkania.Ulica.CompareTo(other.AdresZamieszkania.Ulica);
}
```

Listing 2.11: Ciało metody `CompareTo` interfejsu `IComparable<Pracownik>` porównującej właściwość `Ulica` dwóch obiektów .

W kodzie tym, na rzecz właściwości `AdresZamieszkania.Ulica` (typ `string`) jednego z porównywanych obiektów (`this`) zostaje wywołana metoda `CompareTo`, której argumentem jest również właściwość `AdresZamieszkania.Ulica`, ale drugiego obiektu (`other`). W takim wypadku, pełna deklaracja metody jest następująca:

```
int string.CompareTo(string str);
```

Nie ma w tym zapisie nic dziwnego. Ale gdyby w programie istniała potrzeba posortowania pracowników po np. roku urodzenia, wówczas należałoby wywołać tę samą metodę na rzecz właściwości `DataUrodzenia.Rok`. Wtedy definicja metody `CompareTo()` w klasie `Pracownik` miałaby postać taką jak ta przedstawiona na Listingu 2.12.

```
public int CompareTo(Pracownik other)
{
   return this.DataUrodzenia.Rok.CompareTo(other.DataUrodzenia.Rok);
}
```

Listing 2.12: Definicja metody `CompareTo` interfejsu `IComparable<Pracownik>` porównująca właściwość `Rok` dwóch obiektów.

Zapis jest praktycznie identyczny jak poprzednio, z tym że teraz metoda `CompareTo` jest wywołana na rzecz właściwości `DataUrodzenia.Rok`, a ona zwraca typ `int`. Argumentem tej metody jest również typ `int`. W tym przypadku, pełna deklaracja metody jest następująca:

```
int int.CompareTo(int value);
```

Jest to wywołanie tej samej metody na rzecz innego typu i z innym argumentem. Ale żaden z typów (`string` oraz `int`) nie posiada predefiniowanej metody `CompareTo`. Zatem jak to się dzieje? Otóż za każdym razem dochodzi do automatycznego rozpoznania typu dzięki abstrakcyjnej klasie `AutomationIdentifier`. Potrafi ona zidentyfikować różnego rodzaju typy, zdarzenia, właściwości oraz wzorce. Warto również zdawać sobie sprawę z tego co robi metoda `CompareTo`. Przykładowo, dla porównywanych obiektów typu `string`, zwraca liczbę całkowitą opisującą leksykalny związek pomiędzy obiektami. Porównując znak po znaku w każdym *stringu*, zwraca:

- wartość mniejszą od zera jeżeli *string* danego obiektu jest mniejszy od *stringu* umieszczonego w argumencie,
- wartość równą zero jeżeli porównywane *stringi* są identyczne,
- wartość większą od zera jeżeli *string* danego obiektu jest większy od *stringu* umieszczonego w argumencie.

Sortowanie pracowników przy użyciu metody `Sort(IComparer)`

Omówiony w poprzednim podrozdziale sposób sortowania pracowników listy wymaga, aby klasa `Pracownik` implementowała interfejs `IComparable<Pracownik>`. W metodzie `CompareTo` wystarczy jedynie podać sposób porównywania argumentu metody z referencją `this`, a instrukcja `lista.Sort()` w metodzie `Sortuj` klasy `Lista` przeprowadzi proces sortowania.
Takie rozwiązanie ma niestety jedną wadę: metoda `CompareTo` interfejsu `IComparable` zdefiniowana w klasie `Pracownik` daje możliwość sortowania pracowników wyłącznie po jednej z możliwych składowych klasy. Jest to dość spore ograniczenie, ponieważ proces sortowania powinien umożliwiać wybór, czy sortować pracowników po imieniu, nazwisku, dacie urodzenia, adresie zamieszkania lub zawodzie. Problem ten można jednak w bardzo prosty sposób rozwiązać: wystarczy stworzyć kilka klas i zobligować każdą, za pomocą odpowiedniego interfejsu, do porównania dwóch obiektów. Do tego celu najlepiej wykorzystać interfejs `IComparer<Pracownik>` gdyż, poprzez metodę `Compare`, umożliwia on porównywanie właśnie dwóch obiektów. Wszystkie klasy służące do porównywania obiektów warto dodatkowo umieścić w odrębnej klasie kojarzącej się z procesem sortowania. Listing 2.13 ilustruje omawiane rozwiązanie.

```
public class Sortowanie
{
   public class PoNazwisku : IComparer<Pracownik>
   {
      public int Compare(Pracownik x, Pracownik y)
      {
         return x.Nazwisko.CompareTo(x.Nazwisko);
      }
   }
   public class PoZawodzie : IComparer<Pracownik>
   {
      public int Compare(Pracownik x, Pracownik y)
      {
         return x.Zawod.CompareTo(y.Zawod);
      }
   }
   //Pozostałe opcje sortowania
}
```

Listing 2.13: Nowa klasa `Sortowanie` ze składowymi klasami `PoNzawisku` i `PoZawodzie` implementującymi interfejs `IComparer<Pracownik>`.

Jak można zaobserwować, do celów sortowania stworzona została odrębna klasa o nazwie `Sortowanie` (należy ją umieścić w nowym pliku, np.: **Sortowanie.cs** i dodać do projektu **BibiotekaKlas**). Oznaczona jest ona modyfikatorem `public`, ponieważ wewnątrz tej klasy zdefiniowane są klasy dodatkowe (`PoNazwisku` i `PoZawodzie`), które odpowiedzialne są za porównywanie odpowiednich składowych obiektów – są to właściwości `Nazwisko` oraz `Zawod`. Zarówno klasa `PoNazwisku` jak i `PoZawodzie` implementuje interfejs `IComparer<Pracownik>`, który udostępnia metodę `Compare`. Metoda ta posiada dwa argumenty typu `Pracownik` (`x`, `y`), przez co istnieje możliwość porównania wybranych właściwości.
Aby przeprowadzić proces sortowania wystarczy teraz:

- stworzyć egzemplarz interfejsu `IComparer<Pracownik>`,
- przypisać mu adres instancji typu, po którym porównywać obiekty,
- wywołać metodę `Sort` z argumentem obiektu interfejsu.

Dwie pierwsze linijki Listingu 2.14 prezentują sposób tworzenia obiektu interfejsu i przypisanie mu adresu instancji `PoNazwisku`. Następnie, na rzecz obiektu typu `Lista` wywołana jest metoda `Sortuj`, której argumentem jest egzemplarz interfejsu.

```
IComparer<Pracownik> ic;
ic = new Sortowanie.PoNazwisku(); //Sortowanie po nazwisku
Lista lista = new Lista();
lista.Sortuj(ic);
ic = new Sortowanie.PoZawodzie(); //Sortowanie po zawodzie
lista.Sortuj(ic);
```

Listing 2.14: Ustawienie obiektu interfejsu na odpowiedni typ w celu wyboru sposobu sortowania.

W związku z tym, że metoda `Sortuj` nie została do tej pory zdefiniowana, konieczne jest w tym momencie podanie jej definicji. Na Listingu 2.15 zaprezentowano definicję tej metody. Jest ona składową klasy `Lista`.

```
public void Sortuj(IComparer<Pracownik> ic)
{
   lista.Sort(ic);
}
```

Listing 2.15: Definicja metody `Sortuj` umożliwiającej sortowanie za pomocą obiektu interfejsu `IComparer<Pracownik>`.

2.5 Implementacja interfejsu `IEnumerable` dla klasy `Lista`

Stworzenie obiektu klasy `Lista` umożliwia między innymi dodawanie obiektów typu `Pracownik` do kolekcji. Poruszanie się po tej kolekcji wymaga jednak znajomości liczby jej elementów. Ilość tych elementów ukryta jest we właściwości `Rozmiar`. Aby dostać się do danego elementu konieczne jest wykorzystanie indeksera oraz określenie indeksu danego obiektu. Listing 2.16 ilustruje sposób poruszania się po wszystkich elementach kolekcji typu `Lista`. Na rzecz każdego elementu listy (obiekt typu `Pracownik`) wywoływana jest metoda `ToString()`.

```
Lista lista = new Lista();
for (int i = 0; i < lista.Rozmiar; i++)
{
   Console.WriteLine(lista[i].ToString());
}
```

Listing 2.16: Poruszanie się po kolekcji typu `Lista` przy użyciu pętli `for`.

Jak można zauważyć, na Listingu 2.16 przechodzenie po kolejnych elementach listy odbywa się za pomocą pętli `for`. Pętla odwołuje się do obiektu typu `Pracownik` począwszy od indeksu `0` a skończywszy na `Rozmiar-1`.

Sposób poruszania się po kolekcji można również zrealizować przy użyciu pętli `foreach`. Do tego celu konieczne jest jednak spełnienie kilku warunków:

1. Klasa `Lista` musi implementować interfejs `IEnumerable`. Wiąże się z tym zdefiniowanie w klasie metody `GetEnumerator()`.
2. Metoda `GetEnumerator()` klasy `Lista` musi zwracać obiekt klasy implementującej interfejs `IEnumerator`. Definicję metody prezentuje Listing 2.17.

```
public IEnumerator GetEnumerator()
{
   return new PracownikEnum(lista);
}
```

Listing 2.17: Poruszanie się po kolekcji typu `Lista` przy użyciu pętli `for`.

Jak można zaobserwować, typem zwracanym metody jest `IEnumerator`, a sama metoda zwraca nową instancję typu `PracownikEnum`. Ściśle mówiąc, rusza tutaj do pracy konstruktor klasy `PracownikEnum`, którego argumentem jest obiekt `lista`.

3. Należy stworzyć klasę `PracownikEnum` implementującą interfejs `IEnumerator` o następujących składowych:
   - `public List<Pracownik> lista,`
   - `private int pozycja.`
4. Implementacja interfejsu `IEnumerator` poprzez klasę `PracownikEnum` wymaga zdefiniowania metod `MoveNext()` i `Reset()` oraz właściwości `Current`. Listing 2.18 przedstawia definicję klasy `PracownikEnum` wraz z wymaganymi składowymi.

```
public class PracownikEnum : IEnumerator
{
   public List<Pracownik> lista;
   private int pozycja;
   public PracownikEnum(List<Pracownik> lista)
   {
      pozycja = -1;
      this.lista = lista;
   }
   public Object Current
   {
      get { return lista[pozycja]; }
   }
   public bool MoveNext()
   {
      pozycja++;
      if (pozycja < lista.Capacity)
      {
         return true;
      }
      return false;
   }
   public void Reset()
   {
      pozycja = -1;
   }
}
```

Listing 2.18: Definicja klasy `PracownikEnum` implementującej interfejs `IEnumerator`.

Teraz można już wykorzystać pętlę `foreach` dla kolekcji typu `Lista`. Listing 2.19 ilustruje sposób poruszania się po kolekcji pracowników. W każdej pętli na rzecz obiektu typu `Pracownik` wywoływana jest w tym przypadku metoda `ToString()`.

```
Lista lista = new Lista();
foreach (Pracownik p in lista)
{
   Console.WriteLine(p.ToString());
}
```

Listing 2.19: Wykorzystanie pętli `foreach` do przechodzenia po kolekcji typu `Lista`.

2.6 Metoda `Main`

W metodzie głównej `Main` klasy `Program` należy zdefiniować obiekt klasy `Lista` i wywołać wybrane metody na rzecz tego obiektu, które umożliwią dodawanie, usuwanie, wyszukiwanie, sortowanie pracowników na liście.