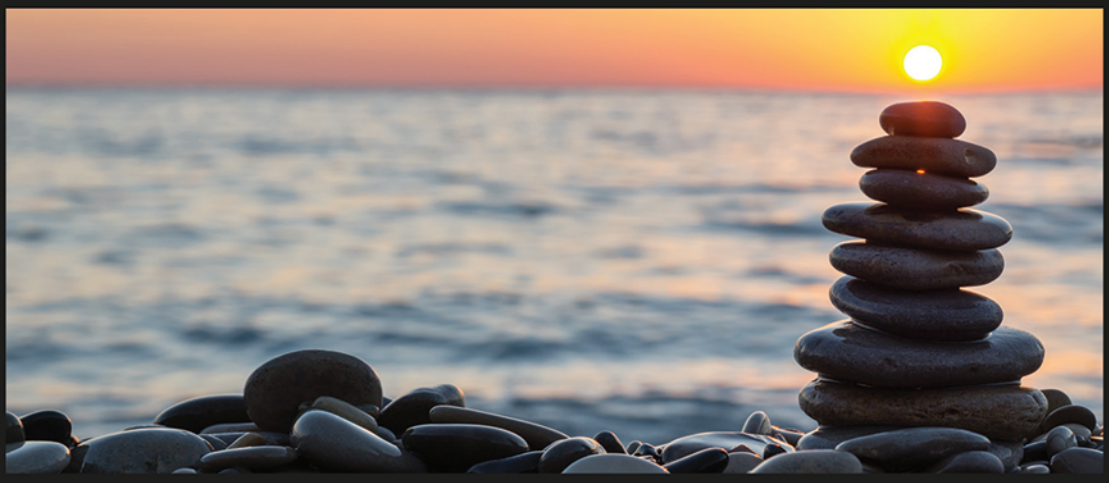
Helion

<packt>



Refaktoryzacja w C#

Jak zredukować dług techniczny i optymalizować kod z Visual Studio, .NET 8 i C# 12

MATT ELAND

Tytuł oryginału: Refactoring with C#: Safely improve .NET applications and pay down technical debt with Visual Studio, .NET 8, and C# 12

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-289-1680-7

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/refawc
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

CZĘŚĆ 2. Bezpieczna refaktoryzacja

CZĘŚĆ 3. Zaawansowana refaktoryzacja przy użyciu sztucznej inteligencji i analizy kodu

Rozdział 3
Refaktoryzacja przepływu sterowania i iteracji

Podczas gdy w pozostałych rozdziałach części I opisuję techniki refaktoryzacji, które można stosować do całych metod i klas, w tym pokazuję sposoby na zwiększenie czytelności i efektywności pojedynczych wierszy kodu.

Programiści najwięcej czasu spędzają na czytaniu kodu wiersz po wierszu i tylko odrobinę poświęcają na jego modyfikowanie. Dlatego nasze wiersze kodu powinny być jak najłatwiejsze w utrzymaniu.

W tym rozdziale opisuję następujące tematy dotyczące ulepszania małych fragmentów kodu:

- Kontrola przepływu sterowania
- Tworzenie obiektów
- Iteracje przez kolekcje
- Refaktoryzacja wyrażeń LINQ
- Przeglądanie i testowanie kodu po refaktoryzacji

Wymagania techniczne

Początkowa wersja kodu do tego rozdziału znajduje się w archiwum pod adresem *https://ftp.helion.pl/przyklady/refawc.zip*, w folderze *Chapter03\Ch3BeginningCode*.

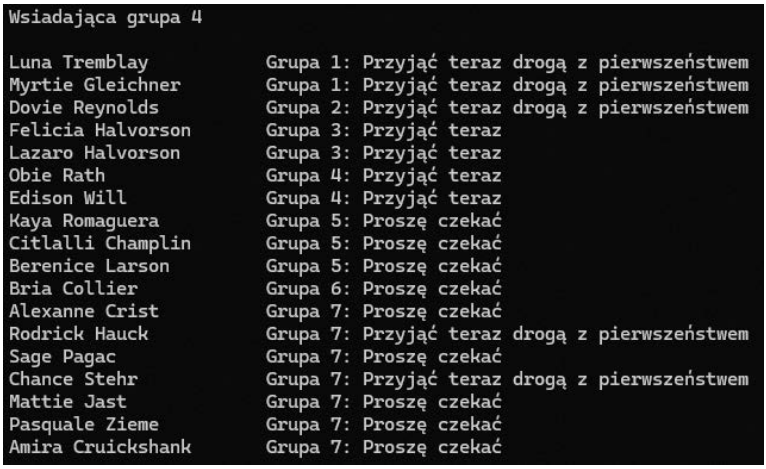
Refaktoryzacja aplikacji do obsługi przyjmowania na pokład

W tym rozdziale opisuję dwie aplikacje linii lotniczych Cloudy Skies:

- Aplikacja **Boarding Status Display** informuje użytkownika, czy powinien już wejść na pokład samolotu, na podstawie tego, jaka grupa obecnie się

melduje, oraz jego biletu, tego, czy pełni czynną służbę wojskową, i tego, czy potrzebuje pomocy, aby dostać się na pokład.

- Aplikacja **Boarding Kiosk** umożliwia pracownikom linii lotniczych obejrzenie listy pasażerów i dostarcza informacji o tym, kto już wszedł na pokład, a kto jeszcze nie. Na rysunku 3.1 widać zrzut ekranu z tej aplikacji.



Rysunek 3.1. Aplikacja Boarding Kiosk

Jako że analizujemy dwie aplikacje naraz, w dalszej części rozdziału prezentuję kod w małych kawałkach. Jeśli jednak chcesz się zorientować w jego ogólnej strukturze, zajdziesz go w całości w pobranych przykładach kodu.

W tym rozdziale pokażę Ci, jak za pomocą drobnych zabiegów refaktoryzacyjnych można poprawić istniejący kod pod względem łatwości utrzymania za pomocą różnych składników języka C#.

Zaczniemy od refaktoryzacji w celu poprawy ogólnego przepływu sterowania.

Kontrola przepływu sterowania

Początkujący programiści zawsze uczą się, jak program wykonuje kolejne wiersze kodu oraz jak za pomocą **instrukcji `if`** i innych składników języka kontrolować to, która instrukcja zostanie wykonana jako następna.

W tym podrozdziale skupimy się na metodzie `CanPassengerBoard` klasy `BoardingProcessor`. Jej początek jest bardzo prosty:

```
public string CanPassengerBoard(Passenger passenger) {
  bool isMilitary = passenger.IsMilitary;
  bool needsHelp = passenger.NeedsHelp;
  int group = passenger.BoardingGroup;
```

Metoda `CanPassengerBoard` pobiera obiekt typu `Passenger` i zwraca łańcuch. Ponadto deklaruje parę zmiennych lokalnych do przechowywania danych z pobranego obiektu.

Zmienne te nie są niezbędne i można się ich pozbyć przez refaktoryzację zmiennych wbudowanych, o której jeszcze będzie mowa w dalszej części tego rozdziału. Ponieważ jednak zwiększają one czytelność kodu znajdującego się dalej, można uznać, że są przydatne. Jest to jeden z powodów, dla których czasami wprowadzamy zmienne lokalne, o czym pisałem w rozdziale 2.

Logika znajdująca się dalej jest już zdecydowanie mniej czytelna, jak widać poniżej:

```
    if (Status != BoardingStatus.PlaneDeparted) {
      if (isMilitary && Status == BoardingStatus.Boarding) {
        return "Przyjąć teraz drogą z pierwszeństwem";
      } else if (needsHelp&&Status==BoardingStatus.Boarding) {
        return "Przyjąć teraz drogą z pierwszeństwem";
      } else if (Status == BoardingStatus.Boarding) {
        if (CurrentBoardingGroup >= group) {
          if (_priorityLaneGroups.Contains(group)) {
            return "Przyjąć teraz drogą z pierwszeństwem";
          } else {
            return "Przyjąć teraz";
          }
        } else {
          return "Proszę czekać";
        }
      } else {
        return "Przyjmowanie nierozpoczęte";
      }
    } else {
      return "Samolot odleciał";
    }
  }
```

W tej metodzie mamy przede wszystkim parę instrukcji `if-else` i kilka porozrzucanych deklaracji metod z umieszczonymi gdzieniegdzie instrukcjami `return`. Mimo że są to fundamentalne konstrukcje programistyczne, trzeba się trochę zastanowić, aby dokładnie zrozumieć działanie tego kodu.

Dla osób, którym nie chce się samodzielnie analizować tej logiki — ten kod działa w następujący sposób:

- Jeśli samolot odleciał, zwraca napis `"Samolot odleciał"`.
- Jeśli pasażerowie jeszcze nie wsiadają do samolotu, zwraca napis `"Przyjmowanie nierozpoczęte"`.
- Jeśli trwa przyjmowanie na pokład samolotu i pasażer potrzebuje pomocy lub jest żołnierzem, zwraca napis `"Przyjąć teraz drogą z pierwszeństwem"`.
- Jeśli trwa przyjmowanie na pokład samolotu i grupa danego pasażera jeszcze nie wchodzi, zwraca napis `"Proszę czekać"`.

- Jeżeli grupa pasażera może wchodzić na pokład, należy mu powiedzieć, aby wszedł normalną drogą lub, jeśli należy do uprzywilejowanej grupy, drogą z pierwszeństwem.

Kod programu jest jednak na tyle zawiły, że wydedukowanie z niego tych zasad zajmuje trochę czasu. Taka złożoność rodzi niepewność i utrudnia innym zrozumienie ogólnego sposobu działania programu.

Te zasady musi zrozumieć każdy, kto będzie pracował z tym kodem. W związku z tym od jego czytelności zależy powodzenie w dłuższej perspektywie.

Odwracanie instrukcji if

Jedną z najprostszych sztuczek pozwalających uprościć skomplikowaną logikę zawierającą zagnieżdżone instrukcje `if` jest ich odwracanie, aby można było wcześniej zwracać wartość.

Obecnie nasza logika wygląda tak:

```
if (Status != BoardingStatus.PlaneDeparted) {
  // Jeszcze 17 wierszy instrukcji if i warunków
} else {
  return "Samolot odleciał";
}
```

Zanim programista czytający kod dojdzie do instrukcji `else` dotyczącej odlotu, zapomni czego dotyczyła pierwsza instrukcja `if`.

Jako że gałąź `else` w tym przypadku jest bardzo prosta i łatwa do zrozumienia, można odwrócić instrukcję `if` w następujący sposób:

1. Zamienić zawartością bloki `if` i `else`.
2. Odwrócić na przeciwne wyrażenie w instrukcji `if`. Operator `==` należy zamienić na `!=` i odwrotnie. Jeśli w użyciu są operatory > lub <, to również należy je zamienić i przy okazji powinno się uwzględnić ewentualny znak równości, tzn. >= zamieniamy na <, a <= na >.

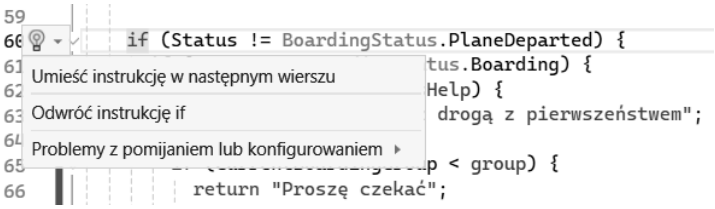
W naszym programie mamy warunek `Status != BoardingStatus.PlaneDeparted`, a więc zmienimy `!=` na `==`, w wyniku czego otrzymamy to:

```
Status == BoardingStatus.PlaneDeparted
```

Te czynności nie zmienią obecnego zachowania programu, ale zmienią kolejność instrukcji w kodzie. Dzięki temu stanie się on bardziej czytelny.

Jeśli wydaje Ci się to skomplikowane, to nie przejmuj się. Visual Studio ma w menu *Szybkie akcje* opcję refaktoryzacji o nazwie *Odwróć instrukcję if*, którą widać na rysunku 3.2.

Wykonanie tej szybkiej akcji zmienia naszą logikę do następującej postaci:



Rysunek 3.2. Szybka akcja refaktoryzacji Odwróć instrukcję if

```
if (Status == BoardingStatus.PlaneDeparted) {
  return "Samolot odleciał";
} else {
  // Jeszcze 17 wierszy instrukcji if i warunków
}
```

Choć już teraz kod jest czytelniejszy, ponieważ nie trzeba pamiętać, czego dotyczy instrukcja `if`, po przeanalizowaniu 17 kolejnych wierszy kodu, nadal można go udoskonalić.

Opuszczanie instrukcji else po instrukcjach return

Jako że instrukcja `return` zawsze powoduje natychmiastowe wyjście z metody, nigdy nie *trzeba* po nim stosować klauzuli `else`, ponieważ wiadomo, że jeśli instrukcja `return` zostanie wykonana, to logika po bloku `if` i tak zostanie pominięta.

To oznacza, że możemy pozbyć się słowa kluczowego `else` i klamry. Następnie usuwamy wcięcie, które było dodane ze względu na blok `else`.

Instrukcję `if` pozostawiamy bez zmian:

```
if (Status == BoardingStatus.PlaneDeparted) {
  return "Samolot odleciał";
}
```

Kod znajdujący się w tej instrukcji `if` jest na tym samym poziomie wcięcia, co ona, a także jest czytelniejszy i łatwiejszy do zrozumienia:

```
if (isMilitary && Status == BoardingStatus.Boarding) {
  return "Przyjąć teraz drogą z pierwszeństwem";
} else if (needsHelp&&Status == BoardingStatus.Boarding) {
  return "Przyjąć teraz drogą z pierwszeństwem";
} else if (Status == BoardingStatus.Boarding) {
  if (CurrentBoardingGroup >= group) {
     if (_priorityLaneGroups.Contains(group)) {
        return "Przyjąć teraz drogą z pierwszeństwem";
     } else {
        return "Przyjąć teraz";
     }
```

```
    } else {
       return "Proszę czekać";
    }
  } else {
    return "Przyjmowanie nierozpoczęte";
  }
```

Jeśli chcemy, tę refaktoryzację możemy wykonać kilka razy, ponieważ nasz kod zawiera kilka konstrukcji `if-return-else`.

Na razie jednak to zostawię, ponieważ teraz chciałbym pokazać inną opcję refaktoryzacji, która może być przydatna.

Restrukturyzacja instrukcji if

W poprzednim fragmencie kodu można znaleźć pewien powtarzający się fragment:

```
if (isMilitary && Status == BoardingStatus.Boarding) {
  return "Przyjąć teraz drogą z pierwszeństwem";
} else if (needsHelp && Status == BoardingStatus.Boarding) {
  return "Przyjąć teraz drogą z pierwszeństwem";
} else if (Status == BoardingStatus.Boarding) {
  // Fragment pominięty dla uproszczenia
} else {
  return "Przyjmowanie nierozpoczęte";
}
```

Mamy tu łańcuch konstrukcji `if-else`, w którym trzy razy z różnych powodów sprawdzamy, czy obecnie trwa przyjmowanie pasażerów na pokład samolotu. Mimo że każda z tych instrukcji `if` różni się od pozostałych, we wszystkich można znaleźć taki sam fragment, który nasuwa mi pytanie, czy dałoby się uniknąć tych powtórzeń.

Pierwszą opcją, jaka przyjdzie nam do głowy, może być *wprowadzenie zmiennej lokalnej* w sposób pokazany w rozdziale 2.:

```
bool isBoarding = Status == BoardingStatus.Boarding;
if (isMilitary && isBoarding) {
  return "Przyjąć teraz drogą z pierwszeństwem";
} else if (needsHelp && isBoarding) {
  return "Przyjąć teraz drogą z pierwszeństwem";
} else if (isBoarding) {
  // Fragment pominięty dla uproszczenia
} else {
  return "Przyjmowanie nierozpoczęte";
}
```

Ten kod jest bardziej czytelny, mimo że dodaliśmy jeden wiersz zawierający definicję nowej zmiennej lokalnej. Jednak tym razem zastosujemy nieco inne podejście.

Zamiast wprowadzać zmienną, poprzestawiamy instrukcje `if` tak, aby zyskać dodatkową warstwę zagnieżdżenia:

```
if (Status == BoardingStatus.Boarding) {
  if (isMilitary) {
     return "Przyjąć teraz drogą z pierwszeństwem";
  } else if (needsHelp) {
     return "Przyjąć teraz drogą z pierwszeństwem";
  } else {
    // Fragment pominięty dla uproszczenia
  }
} else {
  return "Przyjmowanie nierozpoczęte";
}
```

Wyciągnięcie wspólnego warunku z zestawu instrukcji `if` do nadrzędnej instrukcji `if` poprawiło przejrzystość kodu, choć za cenę dodatkowego stopnia zagnieżdżenia.

To uproszczenie pozwoliło jednak dostrzec parę innych możliwości refaktoryzacji, na przykład to, że można połączyć warunki `isMilitary` i `needsHelp`, ponieważ w obu tych przypadkach zwracana jest ta sama wartość, jeśli którykolwiek z nich jest spełniony:

```
if (isMilitary || needsHelp) {
  return "Przyjąć teraz drogą z pierwszeństwem";
}
```

Teraz możemy usunąć klauzulę `else` znajdującą się za konstrukcją `if-return`, aby pozbyć się jednego poziomu wcięcia, pozostawiając tylko logikę grupy przyjmowanej na pokład:

```
if (CurrentBoardingGroup >= group) {
  if (_priorityLaneGroups.Contains(group)) {
     return "Przyjąć teraz drogą z pierwszeństwem";
  } else {
     return "Przyjąć teraz";
  }
} else {
  return "Proszę czekać";
}
```

To wygląda na kolejne miejsce, w którym możemy zastosować odwrócenie instrukcji `if`, aby pozbyć się klauzuli `else` w celu dalszego uproszczenia kodu. Pamiętaj, że musimy zmienić operator >= na <:

```
if (CurrentBoardingGroup < group) {
  return "Proszę czekać";
}
if (_priorityLaneGroups.Contains(group)) {
  return "Przyjąć teraz drogą z pierwszeństwem";
} else {
  return "Przyjąć teraz";
}
```

Jak widać, w miarę upraszczania kod staje się znacznie bardziej czytelny.

Spójrzmy na naszą obecną logikę warunkową po zastosowaniu dotychczasowych zabiegów refaktoryzacyjnych z szerszej perspektywy:

```
if (Status == BoardingStatus.PlaneDeparted) {
  return "Samolot odleciał";
}
if (Status == BoardingStatus.Boarding) {
  if (isMilitary || needsHelp) {
     return "Przyjąć teraz drogą z pierwszeństwem";
  }
  if (CurrentBoardingGroup < group) {
     return "Proszę czekać";
  }
  if (_priorityLaneGroups.Contains(group)) {
     return "Przyjąć teraz drogą z pierwszeństwem";
  } else {
     return "Przyjąć teraz";
  }
} else {
  return "Przyjmowanie nierozpoczęte";
}
```

Ten kod jest czytelniejszy i bardziej odporny na błędy interpretacji. Moglibyśmy jeszcze odwrócić sprawdzanie statusu pokładowego, aby wcześniej zwracać wartość, ale wrócimy do tego później i zrobimy tu coś innego.

Teraz jeszcze bardziej zmniejszymy liczbę wierszy kodu przez zastosowanie zwięzłej konstrukcji języka o nazwie **operator trójargumentowy**.

Operator trójargumentowy

Miłośnicy operatora trójargumentowego być może dostrzegli możliwość użycia go w kodzie, który właśnie refaktoryzujemy.

Jeśli nie znasz tego operatora albo nie czujesz się z nim komfortowo, potraktuj go jako zwięzłe wyrażenie instrukcji typu „jeżeli mój warunek jest spełniony, to użyj tej wartości, a jeżeli nie jest spełniony, to użyj tamtej wartości”.

Składnia operatora trójargumentowego jest następująca: `wyrażenieLogiczne ? wartość ↪GdyPrawda : wartośćGdyFałsz;`.

Innymi słowy, kod bez użycia operatora argumentowego może wyglądać tak:

```
int wartość;
if (jakiśWarunek) {
  wartość = 1;
} else {
  wartość = 2;
}
```

Natomiast z użyciem operatora trójargumentowego to samo można wyrazić w jednym wierszu kodu:

```
int value = jakiśWarunek ? 1 : 2;
```

Jak widać, operator ten pozwala zredukować sześć wierszy kodu do jednego. To właśnie ta zwięzłość jest najważniejszą cechą, dla której programiści stosują w swoim kodzie operator trójargumentowy.

Osoby, które nie przepadają za tym operatorem, twierdzą, że jest mało czytelny — zwłaszcza gdy ktoś chce szybko przejrzeć kod. Inaczej mówiąc, choć operator ten czyni kod bardziej zwięzłym, może on spowalniać programistę w dłuższej perspektywie przez to, że kod będzie trudniejszy w utrzymaniu.

Zobaczmy, jak możemy zastosować operator trójargumentowy do poniższego fragmentu kodu z naszego programu:

```
if (CurrentBoardingGroup < group) {
   return "Proszę czekać";
}
if (_priorityLaneGroups.Contains(group)) {
   return "Przyjąć teraz drogą z pierwszeństwem";
} else {
   return "Przyjąć teraz";
}
```

W tym miejscu sprawdzamy, czy obecnie przyjmowana na pokład grupa ma pierwszeństwo, a następnie nakazujemy użytkownikowi przyjęcie pasażerów z pierwszeństwem lub normalnie, na podstawie wyniku wywołania metody `Contains`.

Ponieważ zwracamy jedną wartość na podstawie wyniku wyrażenia logicznego, możemy zastosować operator trójargumentowy:

```
if (CurrentBoardingGroup < group) {
  return "Please Wait";
}
return _priorityLaneGroups.Contains(group)
        ? "Przyjąć teraz drogą z pierwszeństwem"
        : "Przyjąć teraz";
```

W ten sposób pięć wierszy kodu można zamienić na trzy lub jeden, jeśli wszystkie segmenty konstrukcji umieścimy w tym samym wierszu, co wyrażenie logiczne.

Być może udało Ci się dostrzec, że dzięki tej czynności w bloku kodu pojawiła się kolejna okazja do zastosowania operatora trójargumentowego w odniesieniu do grupy przyjmowanej na pokład — zwrócenie `"Proszę czekać"`, jeśli warunek jest spełniony, lub zwrócenie wyniku poprzedniego operatora trójargumentowego, jeśli warunek nie jest spełniony:

```
return (CurrentBoardingGroup < group)
  ? "Proszę czekać"
  : _priorityLaneGroups.Contains(group)
    ? "Przyjąć teraz drogą z pierwszeństwem"
    : "Przyjąć teraz";
```

Choć język C# dopuszcza takie konstrukcje, to mogę przysiąc, że gdyby współpracownik pokazał mi coś takiego podczas przeglądu kodu, to usłyszałby ode mnie parę naprawdę cierpkich słów!

Wskazówka

Pamiętaj: mniej wierszy kodu nie zawsze jest równoznaczne z większą czytelnością.

Sam zazwyczaj wolę unikać operatora trójargumentowego i nigdy nie tworzę z niego takich łańcuszków, jak pokazałem powyżej. Niemniej jednak czasami zdarza mi się go użyć, kiedy uznam, że jest to korzystne w danym przypadku.

Na przykład czasami metoda jest bardzo prosta i za pomocą wyrażenia trójargumentowego można ją skondensować do postaci jednego wiersza kodu. Ta zmiana daje możliwość użycia składowych z wyrażeniem w treści, które opisałem w rozdziale 4.

Kiedy używam operatora trójargumentowego, każde jego wyrażenie umieszczam w osobnym wierszu, jak pokazałem wcześniej. Pierwszy wiersz zawiera wyrażenie logiczne. Drugi zawiera operator `?` i wartość, która ma zostać użyta, jeśli warunek wyrażenia jest spełniony. Trzeci wiersz zawiera operator `:` i wartość, która ma zostać użyta, jeśli warunek wyrażenia nie jest spełniony.

```
var myVar = booleanExpression
                  ? valueIfTrue
                  : valueIfFalse;
```

Taki format stanowi dla mnie złoty środek między korzyściami płynącymi z użycia bardziej zwięzłego kodu a pogorszeniem jego czytelności i utrudnieniem szybkiego przeglądu.

Zamiana instrukcji if na instrukcje switch

Teraz logika naszej metody jest o wiele bardziej zrozumiała. Teraz wyraźnie widać, że wykonujemy jedną z trzech czynności zależnie od obecnego statusu pokładowego:

- Poinformowanie użytkownika, że samolot już odleciał, jeśli status to `PlaneDeparted`.
- Sprawdzenie, czy pasażer jest żołnierzem, czy potrzebuje pomocy przy wejściu na pokład oraz do jakiej należy grupy pokładowej, w celu określenia statusu pokładowego.
- Powiadomienie użytkownika, że przyjmowanie na pokład jeszcze się nie rozpoczęło, w przypadku pozostałych statusów (obecnie jedynym innym statusem jest `NotStarted`).

Podczas pracy z wartościami wyliczeniowymi często stosuje się ten rodzaj rozgałęzień.

W naszym przypadku wartość `enum` ma tylko trzy stany:

BoardingStatus.cs

```
public enum BoardingStatus {
  NotStarted = 0,
  Boarding = 1,
```

```
    PlaneDeparted = 2,
}
```

Kiedy trzeba sprawdzać jedną zmienną wiele razy pod kątem kilku różnych wartości, to zazwyczaj zamiast instrukcji `if` można użyć **instrukcji `switch`**.

Instrukcje `switch` to w zasadzie uproszczone serie konstrukcji typu `if/else-if/else`, które sprawdzają wartość tej samej zmiennej, tak jak nasz kod sprawdza wartość zmiennej `Status`. Za chwilę zobaczysz przykład użycia instrukcji `switch`, a jeśli jej nie znasz, to możesz sobie ją wyobrazić jako inny sposób zapisu powiązanych ze sobą instrukcji `if/else-if`.

Modyfikację kodu można przeprowadzić własnoręcznie lub przy użyciu specjalnego narzędzia do refaktoryzacji środowiska Visual Studio, jeśli kod ma strukturę typu `if/else-if/else`, jak poniżej:

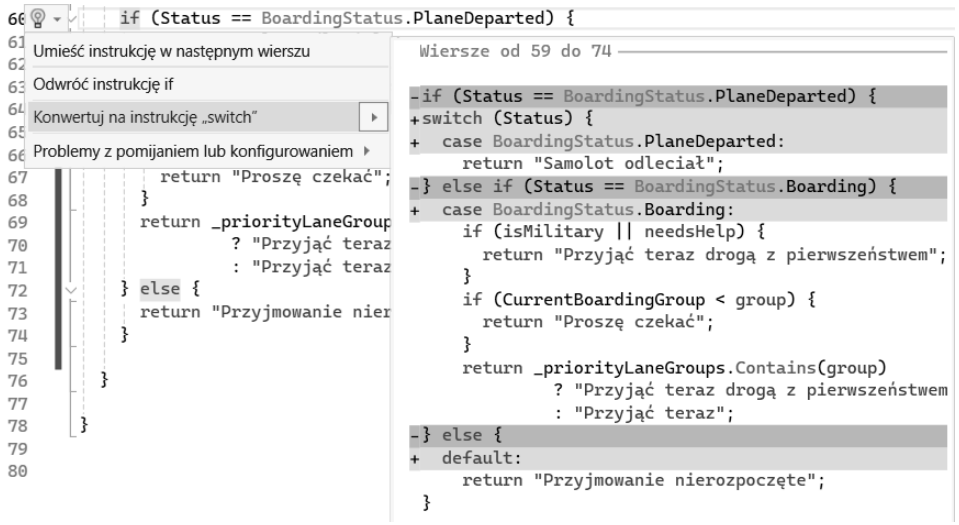
```
if (Status == BoardingStatus.PlaneDeparted) {
  return "Samolot odleciał";
} else if (Status == BoardingStatus.Boarding) {
  if (isMilitary || needsHelp) {
     return "Przyjąć teraz drogą z pierwszeństwem";
  }
  if (CurrentBoardingGroup < group) {
     return "Proszę czekać";
  }
  return _priorityLaneGroups.Contains(group)
           ? "Przyjąć teraz drogą z pierwszeństwem"
           : "Przyjąć teraz";
} else {
  return "Przyjmowanie nierozpoczęte";
}
```

W tym kodzie zwróć uwagę na dodatek słowa kluczowego `else` (wytłuszczone na listingu), aby powstała konstrukcja `if/else-if/else` dająca Visual Studio możliwość rozpoznania rodzaju refaktoryzacji, którą chcemy zastosować.

Kiedy nasz kod odpowiada temu wzorcowi, to po zaznaczeniu instrukcji `if` w menu *Szybkie akcje* pojawi się opcja refaktoryzacji *Konwertuj na instrukcję „switch”*, jak widać na rysunku 3.3.

Dzięki tej zamianie nasza logika oparta na statusie staje się o wiele bardziej przejrzysta:

```
switch (Status) {
  case BoardingStatus.PlaneDeparted:
     return "Samolot odleciał";
  case BoardingStatus.Boarding:
     if (isMilitary || needsHelp) {
        return "Przyjąć teraz drogą z pierwszeństwem";
     }
     if (CurrentBoardingGroup < group) {
        return "Proszę czekać";
     }
```



Rysunek 3.3. Opcja refaktoryzacji Konwertuj na instrukcję „switch"

```
      return _priorityLaneGroups.Contains(group)
               ? "Przyjąć teraz drogą z pierwszeństwem"
               : "Przyjąć teraz";
    default:
      return "Przyjmowanie nierozpoczęte";
  }
```

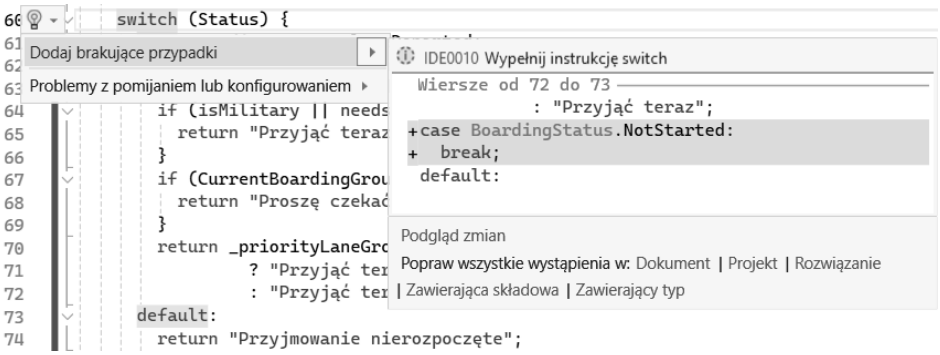
Dla mnie jako osoby czytającej ten kod przeglądanie i interpretowanie takich konstrukcji jest o wiele łatwiejsze niż łańcuchów `if/else-if/else`, mimo że logika cały czas jest taka sama. W konstrukcji `if/else-if/else`, jeśli się przyjrzę, mogę zauważyć, że wartość jednej zmiennej jest sprawdzana kilka razy. Natomiast w instrukcji `switch` jest to wyraźnie widoczne.

Inną zaletą instrukcji `switch` jest to, że umożliwia zastosowanie wbudowanej opcji refaktoryzacji, gdy sprawdzana jest wartość wyliczenia (taka jak `BoardingStatus`) i brakuje klauzuli `case` dla jednej lub większej liczby wartości tego wyliczenia.

Ta opcja pojawia się w menu *Szybkie akcje* dla instrukcji `switch` pod nazwą *Dodaj brakujące przypadki*, jak widać na rysunku 3.4.

W naszym przypadku status `NotStarted` możemy połączyć z przypadkiem domyślnym, aby uzyskać bardziej oczywistą listę opcji:

```
switch (Status) {
  case BoardingStatus.PlaneDeparted:
     return "Samolot odleciał";
  case BoardingStatus.Boarding:
     if (isMilitary || needsHelp) {
        return "Przyjąć teraz drogą z pierwszeństwem";
     }
     if (CurrentBoardingGroup < group) {
```



Rysunek 3.4. Opcja refaktoryzacji Dodaj brakujące przypadki w menu Szybkie akcje

Ostrzeżenie

Ostrzegam, że opcja refaktoryzacji *Dodaj brakujące przypadki* w tym przypadku może spowodować zmianę zachowania. Jej wbudowana w środowisko implementacja dodaje status `NotStarted` i powoduje wyjście z instrukcji `switch`, zamiast zwracać wartość — wcześniej była zwracana wartość przez słowo kluczowe `default`.

Kompilator C# zaznaczy ten błąd, ponieważ metoda nie będzie zwracać wartości na tej ścieżce. Niemniej jednak dodanie brakujących przypadków, kiedy w instrukcji `switch` znajduje się klauzula `default`, zazwyczaj powoduje zmianę zachowania.

```
        return "Proszę czekać";
    }
    return _priorityLaneGroups.Contains(group)
             ? "Przyjąć teraz drogą z pierwszeństwem"
             : "Przyjąć teraz";
  case BoardingStatus.NotStarted:
  default:
    return "Przyjmowanie nierozpoczęte";
}
```

Ten kod jest o wiele czytelniejszy niż jego poprzednia wersja i dodatkowo przepływ logiki na podstawie statusu stał się oczywisty.

W rzeczywistej aplikacji mógłbym zmienić domyślny przypadek tak, aby był w nim zgłaszany wyjątek, co by mnie wprost informowało, że dany status nie jest obsługiwany. Wyglądałoby to mniej więcej tak:

```
case BoardingStatus.NotStarted:
  return "Przyjmowanie nierozpoczęte";
default:
  throw new NotSupportedException($"Nieobsługiwany status: {Status}");
```

Mogłoby mnie też kusić, aby *wyodrębnić metodę* — jak pokazałem w rozdziale 2. — i przenieść logikę obsługi statusu przyjmowania na pokład do osobnej metody. Powstrzymam się jednak od tego, żeby móc zaprezentować sposób użycia wyrażeń `switch`.

Konwersja na wyrażenia switch

Wyrażenia `switch` to rozwinięta wersja instrukcji `switch`, która wykorzystuje wyrażenia **dopasowywania wzorców** w celu uproszczenia i rozszerzenia możliwości tradycyjnych instrukcji `switch`.

Wyrażenia `switch` to dość nowy dodatek do języka C#, wprowadzony w wersji C# 8 w 2019 roku. Choć jest więc dostępny już od paru lat, to wciąż jest na tyle nowy, że wielu programistów go nie zna albo nie ma doświadczenia w jego używaniu.

Proste wyrażenie `switch` jest bardzo podobne do instrukcji `switch`:

```
return Status switch {
  BoardingStatus.PlaneDeparted => "Samolot odleciał",
  BoardingStatus.NotStarted => "Przyjmowanie nierozpoczęte",
  BoardingStatus.Boarding => "Przyjąć teraz",
  _ => "Jakiś inny status",
};
```

Wyrażenia `switch` wyglądają bardzo podobnie do instrukcji `switch`, ale różnią się od nich w następujących aspektach:

- Na początku mają wartość do sprawdzenia, po której znajduje się słowo kluczowe `switch`. Instrukcja ma postać `switch` (*`wartość`*).
- Nie używa się w nich słów kluczowych `case` ani `break`.
- Poszczególne przypadki składają się ze strzałki `=>`, która po lewej stronie ma warunek, a po prawej — wartość do użycia, jeśli ten warunek będzie spełniony.
- Zamiast słowa kluczowego `default` jest znak _ oznaczający jakiekolwiek inne dopasowanie.

Jedną z zalet wyrażeń `switch` jest to, że są bardzo zwięzłe, a jednocześnie dość czytelne. Należy jednak pamiętać, że możliwości tych wyrażeń wykraczają poza to, co do tej pory omówiłem.

Być może udało Ci się zauważyć, że przykładowe wyrażenie `switch`, które pokazałem powyżej, nie obsługuje poprawnie logiki przyjmowania pasażerów na pokład. Mówiąc konkretnie, wcześniej mieliśmy zasady dotyczące przyjmowania żołnierzy, osób potrzebujących pomocy, grup i dróg dla osób uprzywilejowanych, a nic z tego nie zostało uwzględnione w opisywanym bloku kodu.

Teraz więc spójrz na wyrażenie `switch`, które obsługuje wszystkie te przypadki:

```
return Status switch {
  BoardingStatus.PlaneDeparted => "Samolot odleciał",
  BoardingStatus.NotStarted => "Przyjmowanie nierozpoczęte",
  BoardingStatus.Boarding when isMilitary || needsHelp
    => "Przyjąć teraz drogą z pierwszeństwem",
  BoardingStatus.Boarding when CurrentBoardingGroup < group
    => "Proszę czekać",
```

```
    BoardingStatus.Boarding when
      _priorityLaneGroups.Contains(group)
        => "Przyjąć teraz drogą z pierwszeństwem",
    BoardingStatus.Boarding => "Przyjąć teraz",
    _ => "Jakiś inny status",
};
```

Ten kod odrobinę się różni od poprzedniego wyrażenia `switch`. Status `Boarding` został powtórzony cztery razy i czasami towarzyszy mu słowo kluczowe `when`.

Ten kod przy użyciu techniki dopasowywania wzorców nie tylko sprawdza, czy `Status` to `Boarding`, ale też czy inne warunki również są spełnione. W efekcie możemy sprawdzać status i opcjonalnie inne wyrażenie logiczne, które znajduje się za słowem kluczowym `when`.

Jeśli w obu przypadkach jest fałsz, to wyrażenie `switch` wykonuje kolejny wiersz kodu. W efekcie wyrażenia `switch` są zbiorami reguł dopasowywania, które sprawdzają, czy pierwsza reguła daje w wyniku prawdę.

Dopasowywanie wzorców

Dopasowywanie wzorców to jeden z nowszych elementów składni języka C#, który umożliwia zwięzłe sprawdzanie różnych właściwości oraz aspektów obiektów i zmiennych. Szerzej na temat składni dopasowywania wzorców piszę w rozdziale 10., a ten punkt można traktować jako podstawowe wprowadzenie do tego tematu.

Innymi słowy, to wyrażenie `switch` sprawdza poniższe warunki i reaguje na pierwszy, który jest spełniony:

1. Samolot odleciał.
2. Jeszcze nie zaczęto wpuszczać pasażerów do samolotu.
3. Rozpoczęto wpuszczanie pasażerów do samolotu i bieżący pasażer jest żołnierzem lub potrzebuje pomocy.
4. Grupa, do której należy pasażer, jeszcze nie została wezwana.
5. Grupa, do której należy pasażer, jest właśnie wpuszczana drogą z pierwszeństwem.
6. Grupa, do której należy pasażer, jest właśnie wpuszczana, ale nie drogą z pierwszeństwem.
7. Inny status.

Wyrażenia `switch` są zwięzłe i łączą strukturalną klarowność instrukcji `switch` z możliwościami dopasowywania wzorców i słowa kluczowego `when`, co pozwala na tworzenie bardzo czytelnych konstrukcji logicznych.

Wyrażenia `switch`, jak każde narzędzie w zasobach programisty, nie są rozwiązaniem każdego możliwego problemu, a poza tym ludzie z Twojego zespołu mogą nie być nimi tak zachwyceni, jak ja. Niemniej jednak warto o nich wiedzieć, ponieważ pozwalają uprościć kod, aby był czytelny oraz łatwy w utrzymaniu i rozwijaniu.

Do składni dopasowywania wzorców jeszcze wrócimy w rozdziale 10. Teraz natomiast przejdziemy do sposobów usprawniania pracy z kolekcjami obiektów.

Tworzenie obiektów

Metoda `CanPassengerBoard` jest już wystarczająco ulepszona, więc możemy przyjrzeć się sposobom **tworzenia obiektów** i wprowadzić parę poprawek, które uproszczą ten proces w naszym kodzie.

> **Uwagi terminologiczne**
>
> Początkujący programiści często potykają się o pewne wyrażenia, które są powszechnie używane przez doświadczonych programistów. Na przykład w tym podrozdziale dużo piszę o tworzeniu obiektów, czyli tworzeniu konkretnych instancji klasy za pomocą słowa kluczowego `new`. Proces ten bywa nazywany **instancjacją**. Jeśli więc kiedyś spotkasz się z tym słowem, wiedz, że chodzi po prostu o tworzenie obiektu klasy.
>
> Kod źródłowy do tego rozdziału mógłby pochodzić z dowolnego źródła, ale w przykładach przedstawiam fragmenty kodu dwóch metod z pliku *PassengerTests.cs*.

Zamiana var na konkretne określenia typów

Pierwszy wiersz kodu, na którym chcę się skupić, pochodzi z testu jednostkowego:

PassengerTests.cs

```
var p = Build(first, last);
```

Celowo pominąłem szerszy kontekst sąsiednich wierszy, aby podkreślić to, o czym chcę powiedzieć: zastanów się, jakiego typu danych jest zmienna `p`.

Zmienna `p` przechowuje wynik działania metody `Build`, która przyjmuje parę parametrów o nazwach `first` i `last`. Na podstawie tego wiersza kodu w żaden sposób nie jesteśmy w stanie z całą pewnością stwierdzić, jaki typ będzie miała ta zmienna.

Jest tak dlatego, że w jej deklaracji użyto słowa kluczowego `var`, które stanowi skrótową formę następującego zapisu: „Hej, kompilatorze, kiedy będziesz kompilować ten kod, określ typ danych tej zmiennej i w skompilowanym kodzie zastąp słowo kluczowe `var` rzeczywistym typem danych”.

Inaczej mówiąc, słowo kluczowe `var` jest zazwyczaj skrótem pozwalającym nie podawać pełnej nazwy typu danych. Jednak ceną za tę wygodę jest utrudnienie zrozumienia kodu tym, którzy będą go czytać, spowodowane brakiem typu zmiennej.

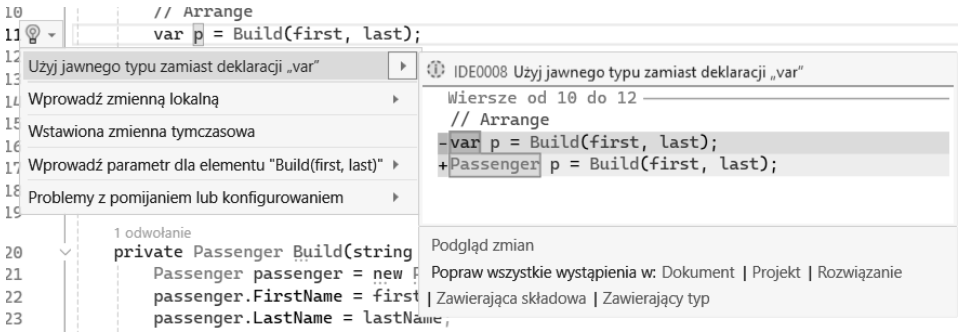
To ma sens, kiedy mamy do czynienia ze złożonym typem danych, takim jak `IDictiona ↪ry<Guid, HashSet<string>>`. Natomiast robi się trochę śmieszne, kiedy tym słowem kluczowym zastępujemy krótkie nazwy typów danych, takie jak `int`.

Inne zastosowania słowa kluczowego var

Słowo kluczowe `var` ma także inne zastosowania. Dzięki niemu można na przykład przechowywać wartości **typów anonimowych** i inne trudne do reprezentowania struktury typowe. Jednak w tej książce skupiam się na najpowszechniejszym sposobie użycia tego słowa w bazach kodu.

W środowisku Visual Studio można najechać wskaźnikiem myszy na deklarację zmiennej, aby podejrzeć jej rzeczywisty typ. W tym przypadku zmienna `p` reprezentuje obiekt typu `Passenger`. To jednak i tak spowalnia czytanie kodu.

Dlatego w takich przypadkach zalecam skorzystanie z opcji refaktoryzacji o nazwie *Użyj jawnego typu zamiast deklaracji „var”* — rysunek 3.5.



Rysunek 3.5. Bezpośrednie określanie typów

Dzięki temu kod jest znacznie bardziej czytelny:

```
Passenger p = Build(first, last);
```

Oczywiście słowo kluczowe `var` udostępniono nie bez powodu, w celu rozwiązania pewnych konkretnych problemów, w tym związanych z powtarzalnością w instrukcjach przypisania. W następnej kolejności przyjrzymy się słowu kluczowemu **`new` z określeniem typu docelowego**, które również pozwala rozwiązać tego typu problemy.

Prostsze tworzenie przy użyciu słowa kluczowego new z określeniem typu docelowego

Jednym z problemów, które ma rozwiązywać słowo kluczowe `var`, jest tworzenie zmiennych w sposób pokazany poniżej:

```
private Passenger Build(string firstName, string lastName){
     Passenger passenger = new Passenger();
     passenger.FirstName = firstName;
     passenger.LastName = lastName;
     return passenger;
}
```

Kiedy tworzymy obiekt klasy `Passenger` i przypisujemy go do nowej zmiennej typu `Passenger`, to powtarzamy się, ponieważ po obu stronach operatora przypisania (`=`) używamy nazwy tej klasy.

Słowo kluczowe `var` pozwoliło uprościć tworzenie obiektów w takich przypadkach przy zachowaniu czytelności kodu: `var passenger = new Passenger();`. Tutaj słowo kluczowe `var` upraszcza lewą stronę instrukcji przypisania przez wyeliminowanie nazwy typu zmiennej.

W C# 9 wprowadzono słowo kluczowe **`new` z określeniem typu docelowego**, które umożliwia uproszczenie prawej strony operatora przypisania przez dodanie informacji, że klasa, której obiekt tworzymy, jest taka sama jak klasa zmiennej będącej celem tego operatora.

Mówiąc inaczej, słowo kluczowe `new` z określeniem typu docelowego pozwala nakazać C# utworzenie obiektu takiego typu, jak typ zmiennej, w której ma ona zostać zapisana. Dzięki temu nie musimy używać słowa kluczowego `var` i unikamy powtórzeń:

```
Passenger passenger = new();
```

Uwielbiam tę składnię i używam jej wszędzie w swoich programach. Inni programiści, którzy widzą ją po raz pierwszy, bywają zaskoczeni, ale to drobna jednorazowa niedogodność, która umożliwia pisanie zwięzłego i jednocześnie czytelnego kodu.

Wskazówka

W menu *Szybkie akcje* środowiska Visual Studio jest dostępna opcja *Użyj operatora „new(…)”*, która umożliwia zamianę tradycyjnej metody tworzenia obiektu na składnię ze słowem kluczowym `new` z określeniem typu docelowego.

Skoro mowa o tworzeniu obiektów, teraz pokażę Ci, jak **inicjalizatory obiektów** mogą pomóc w ustawianiu ich własności podczas ich tworzenia.

Inicjalizatory obiektów

Wrócimy po raz kolejny do metody `Build` z poprzedniego przykładu i przyjrzymy się sposobowi skonfigurowania utworzonego obiektu reprezentującego pasażera:

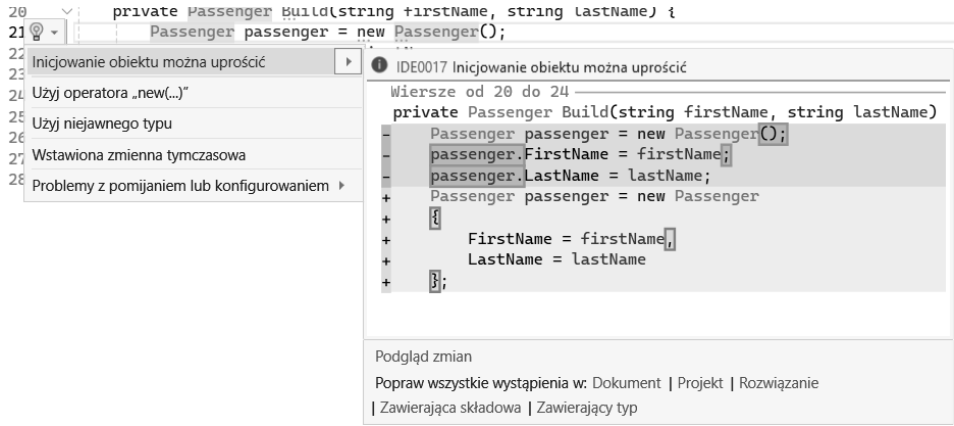
```
private Passenger Build(string firstName, string lastName){
  Passenger passenger = new();
  passenger.FirstName = firstName;
  passenger.LastName = lastName;
  return passenger;
}
```

W tym kodzie nie ma nic złego, ale jest odrobinę powtarzalny.

Mówiąc dokładniej, powtarza się w nim informacja o konfigurowanym obiekcie, czyli element `passenger.` przed każdą własnością, której jest przypisywana wartość.

Kiedy w użyciu są tylko dwie własności, to problem jest minimalny. Ale wyobraź sobie większy obiekt, zawierający 10 albo i więcej własności, które musisz skonfigurować. W takim kodzie byłoby bardzo dużo powtórzeń, które mogłyby odwracać uwagę od nazw własności.

Jednym z rozwiązań tego problemu jest użycie konstruktora przyjmującego parametry reprezentujące wszystkie własności (w dalszej części rozdziału pokazuję taki konstruktor). Innym wyjściem jest użycie inicjalizatora obiektu. Zapewne się domyślasz, że środowisko Visual Studio ma odpowiednią szybką akcję, choć o dość nietypowej nazwie *Inicjowanie obiektu można uprościć* (rysunek 3.6).



Rysunek 3.6. Upraszczanie inicjalizacji obiektu

Zastosowanie tej opcji refaktoryzacji upraszcza nasz kod:

```
private Passenger Build(string firstName, string lastName){
    Passenger passenger = new() {
        FirstName = firstName,
        LastName = lastName
```

```
        };
        return passenger;
    }
```

Uwielbiam tę składnię, która doskonale współpracuje z własnościami `init` i `required`, opisanymi w rozdziale 10. Należy jednak pamiętać, że używanie inicjalizatorów obiektów ma wadę związaną z analizą stosu.

Kiedy w użyciu jest inicjalizator obiektu ustawiający kilka jego własności i w trakcie wykonywania obliczeń wartości przeznaczonej do zapisania następuje wyjątek, to wyjątek ten nie wskazuje, w którym wierszu kodu wystąpił błąd ani która własność miała zostać zaktualizowana. Informuje tylko, że gdzieś w inicjalizatorze coś się nie udało.

Gdyby natomiast poszczególne własności były inicjalizowane w osobnych wierszach, to wyjątek określałby problematyczny wiersz. To oczywiście może być argument za tym, aby unikać wykonywania obliczeń w inicjalizatorach, które mogą generować wyjątki.

Do inicjalizatorów jeszcze wrócimy w rozdziale 10. przy okazji omawiania wyrażeń `init`, `required` i `with`, a teraz poświęcimy trochę czasu kolekcjom.

Iterowanie kolekcji

Aby przyjrzeć się kolekcjom, wrócimy do klasy `BoardingProcessor` i odnajdziemy w niej metodę `DisplayPassengerBoardingStatus`. Przeanalizujemy ją po kawałku, począwszy od sygnatury:

```
public void DisplayBoardingStatus(
  List<Passenger> passengers, bool? hasBoarded = null) {
```

Metoda ta przyjmuje listę obiektów klasy `Passenger` i, opcjonalnie, logiczny parametr `hasBoarded`, który może przyjmować wartości `true`, `false` oraz `null`. Parametr ten służy do filtrowania listy pasażerów na podstawie jego wartości:

- `true`: tylko pasażerowie, którzy weszli na pokład samolotu,
- `false`: tylko pasażerowie, którzy nie weszli na pokład samolotu,
- `null`: nie filtrować według statusu pokładowego (opcja domyślna).

Parametr filtrowania dopuszczający wartość `null` często widuję w metodach wyszukujących i dokładniej opisuję go w rozdziale 5.

Następny fragment kodu metody `DisplayBoardingStatus` odpowiada za logikę filtrowania:

```
List<Passenger> filteredPassengers = new();
for (int i = 0; i < passengers.Count; i++) {
   Passenger p = passengers[i];
   if (!hasBoarded.HasValue || p.HasBoarded==hasBoarded) {
      filteredPassengers.Add(p);
   }
}
```

Na tym fragmencie skupimy teraz uwagę. Tworzy on na podstawie opcji filtrowania wybranej przez użytkownika nową listę pasażerów i robi to przez iterowanie po kolekcji pasażerów `passengers`, z której warunkowo wybiera elementy.

Uwaga terminologiczna

Początkujący programiści czasami mają problem ze zrozumieniem pojęcia **iteracja**. Słowo to oznacza po prostu przeglądanie elementów kolekcji za pomocą pętli.

Pozostała część omawianej metody odpowiada za wyświetlenie pasażerów na monitorze agenta:

```
    DisplayBoardingHeader();
    foreach (Passenger passenger in filteredPassengers) {
       string statusMessage = passenger.HasBoarded
          ? "Na pokładzie"
          : CanPassengerBoard(passenger);
       Console.WriteLine($"{passenger.FullName,-23} Group
          {passenger.BoardingGroup}: {statusMessage}");
    }
}
```

Dla każdego pasażera, którego dane chcemy wyświetlić, drukujemy imię i nazwisko, grupę i wiadomość pokazywaną w aplikacji do obsługi przyjęć na pokład samolotu lub napis „Na pokładzie”, jeśli dana osoba już weszła do samolotu.

Ogólnie rzecz biorąc, jest to prosta metoda. Zawiera mniej niż 20 wierszy kodu, a więc ma rozmiar, który zazwyczaj oznacza łatwe utrzymanie.

Zobaczmy, co można w niej poprawić.

Pętla foreach

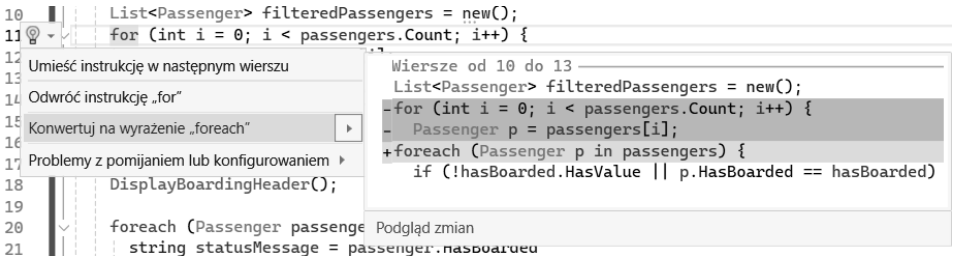
Jeszcze raz spójrz na kod, który filtruje listę pasażerów i wynik zapisuje w nowej liście pasażerów:

```
List<Passenger> filteredPassengers = new();
for (int i = 0; i < passengers.Count; i++) {
  Passenger p = passengers[i];
  if (!hasBoarded.HasValue || p.HasBoarded == hasBoarded) {
     filteredPassengers.Add(p);
  }
}
```

Wprawdzie nie jest to zbyt skomplikowany kod, ale w oczy rzuca mi się użycie pętli `for` do przeglądania listy pasażerów. W treści tej pętli w ogóle nie używamy zmiennej indeksowej `i`, która służy nam tylko do pobierania kolejnych pasażerów z listy według indeksu.

Wszystkie pętle `for` tego typu, czyli takie, które nie robią nic skomplikowanego (na przykład nie zaczynają przeglądania listy od innego miejsca niż początek, nie przeglądają kolekcji wstecz ani nie pomijają żadnych elementów), z reguły można wymienić na pętlę `foreach`.

Aby przekonwertować pętlę `for` na pętlę `foreach`, można ją zaznaczyć i wybrać opcję refaktoryzacji środowiska Visual Studio o nazwie *Konwertuj na wyrażenie „foreach”* (rysunek 3.7).



Rysunek 3.7. Opcja refaktoryzacji Konwertuj na wyrażenie „foreach” w menu Szybkie akcje

To powoduje zamianę pętli na `foreach` i usunięcie deklaracji zmiennej:

```
List<Passenger> filteredPassengers = new();
foreach (Passenger p in passengers) {
  if (!hasBoarded.HasValue || p.HasBoarded == hasBoarded) {
     filteredPassengers.Add(p);
  }
}
```

Używam pętli `foreach`, kiedy tylko mogę, ponieważ pozwala pozbyć się deklaracji zmiennej i indeksatora oraz zwiększa czytelność kodu.

Prawie wszystkie pętle `for` zaczynają od `0` i przeglądają kolekcje do samego końca po jednym elemencie na raz. Jednak niektóre są inne. Dlatego zawsze gdy napotykam pętlę `for`, sprawdzam, czy jest to standardowa konstrukcja, czy może ma jakieś wyjątkowe cechy. W przypadku pętli `foreach` nie muszę tego robić, ponieważ jej składnia nie pozwala na nic innego. To zwiększa komfort i pozwala na szybsze czytanie kodu, który jest prostszy i dzięki temu łatwiejszy w utrzymaniu.

Ponadto pętli `foreach` można używać do pracy ze wszystkim, co implementuje interfejs `IEnumerable`, natomiast pętla `for` wymaga, aby przeglądana kolekcja miała indeksator. Oznacza to, że pętla `foreach` umożliwia przeglądanie większej liczby typów kolekcji niż pętla `for`.

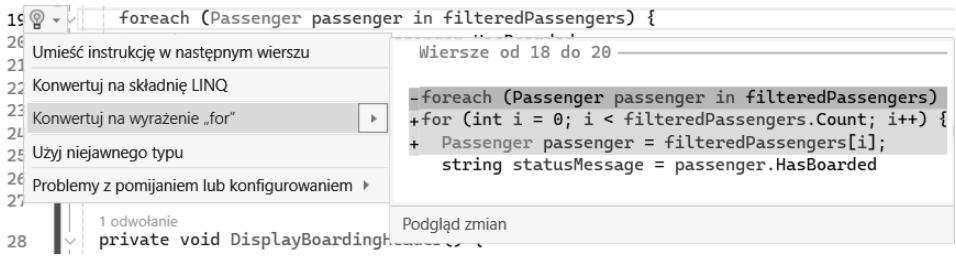
Interfejsy kolekcji

Platforma .NET udostępnia kilka interfejsów kolekcji, takich jak `IEnumerable`, `ICollection`, `IList`, `IReadOnlyList` czy `IReadOnlyCollection`. Znajomość tych typów kolekcji jest przydatna, ale nie wymagana podczas lektury tej książki. W podrozdziale „Dalsza lektura" na końcu tego rozdziału znajduje się odnośnik do dodatkowych informacji na temat tych interfejsów. Na razie wystarczy pamiętać, że interfejs `IEnumerable` to specjalny sposób na oznaczenie czegoś, co można przeglądać za pomocą pętli `foreach`.

Konwersja na pętlę for

Mimo że pętle `foreach` są świetne i zazwyczaj są moim pierwszym wyborem, czasami potrzebna jest pętla `for`, która daje odrobinę większą kontrolę. Jeśli trzeba przejrzeć kolekcję w niestandardowy sposób albo chcemy wykorzystać zmienną indeksową do czegoś innego niż odczyt zmiennej z kolekcji, to najczęściej sięgamy po pętlę `for`.

W Visual Studio dostępna jest opcja refaktoryzacji o nazwie *Konwertuj na wyrażenie „for"*, która zamienia pętle `foreach` na pętle `for` — rysunek 3.8.



Rysunek 3.8. Konwersja pętli foreach na pętlę for

Rzadko używam tej opcji refaktoryzacji, ale czasami się przydaje.

Na razie pozostawimy w kodzie pętlę `foreach` i zobaczymy, jak można go ulepszyć za pomocą technologii LINQ.

Konwersja na LINQ

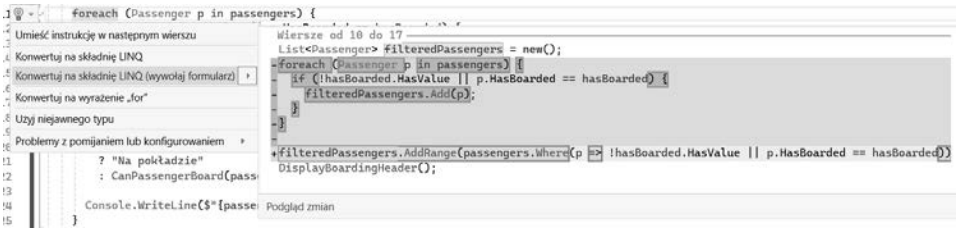
Na rysunku 3.8 można zauważyć, że oprócz opcji konwersji na pętlę `foreach` w menu dostępna jest także opcja konwersji na składnię LINQ.

Nazwa ***LINQ*** to akronim od angielskich słów *Language INtegrated Query*. Jest to zestaw metod rozszerzających, które działają na każdej kolekcji implementującej interfejs `IEnumerable`. Za ich pomocą można szybko wykonywać operacje agregacji, transformacji i filtrowania kolekcji przy użyciu funkcji strzałkowych.

Funkcje strzałkowe

Funkcje strzałkowe (nazywane także wyrażeniami lambda) to zwięzła składnia do reprezentowania krótkich metod oparta na symbolu =>. Zakładam, że czytelnik tej książki ma podstawową wiedzę o tego typu funkcjach. Jeśli potrzebujesz więcej informacji lub chcesz odświeżyć swoje wiadomości, zajrzyj do podrozdziału „Dalsza lektura" na końcu tego rozdziału.

Spójrzmy, co się stanie z naszą pętlą `foreach`, kiedy użyjemy opcji refaktoryzacji *Konwertuj na składnię LINQ (wywołaj formularz)* w menu *Szybkie akcje* tej pętli — rysunek 3.9.



Rysunek 3.9. Konwersja pętli `foreach` na składnię LINQ

Ta czynność spowoduje zamianę naszej pętli `foreach` na krótki fragment kodu:

```
List<Passenger> filteredPassengers = new();
filteredPassengers.AddRange(passengers.Where(p => !hasBoarded.HasValue ||
↪p.HasBoarded == hasBoarded));
```

Ten kod pobiera naszą kolekcję `passengers` i wywołuje metodę rozszerzającą `Where`, która tworzy i zwraca nową, implementującą interfejs `IEnumerable` sekwencję pasażerów, dla których funkcja strzałkowa `p => !hasBoarded.HasValue || p.HasBoarded == hasBoarded` zwraca wartość `true`.

Metody rozszerzające

Metody rozszerzające to zdefiniowane w statycznych klasach metody statyczne, które umożliwiają tworzenie składni wyglądającej tak, jakby dodawała nowe metody do istniejących typów. LINQ w szerokim zakresie wykorzystuje metody rozszerzające związane z różnymi interfejsami. W rozdziale 4. opisuję sposoby ich tworzenia.

Ta czynność nie zmodyfikuje naszej pierwotnej kolekcji, tylko spowoduje utworzenie nowej kolekcji obiektów klasy `Passenger`, które następnie zostaną przekazane do metody `filteredPassengers.AddRange`.

Mimo że ten kod już jest bardzo zwięzły, możemy go jeszcze poprawić przez wykorzystanie konstruktora generycznej klasy `List`.

Klasa `List<T>` ma konstruktor przyjmujący interfejs `IEnumerable<T>` i umożliwiający efektywne utworzenie nowej listy z sekwencji elementów. Dzięki temu możemy uniknąć potrzeby wywoływania metody `AddRange`, co pozwala uprościć nasz kod do postaci pojedynczej instrukcji:

```
List<Passenger> filteredPassengers =
  new(passengers.Where(p => !hasBoarded.HasValue ||
     p.HasBoarded == hasBoarded));
```

Gdybyśmy chcieli, to moglibyśmy także całkowicie pozbyć się zmiennej `filteredPassen` ↪`gers` przez przefiltrowanie listy pasażerów i przypisanie wyniku tego filtrowania z powrotem do tej samej zmiennej:

```
passengers = passengers.Where(p=>!hasBoarded.HasValue ||
      p.HasBoarded==hasBoarded).ToList();
```

Za pomocą metody `Where` wygenerowaliśmy interfejs `IEnumerable<Passenger>` zawierający naszych pasażerów, a następnie wywołaliśmy na nim metodę `ToList`, aby przekonwertować go z powrotem na listę, którą można przechowywać w parametrze `passengers`.

Pamiętaj też, że wszystkie wystąpienia `filteredPassengers` należy teraz zmienić na `passengers`:

```
foreach (Passenger passenger in passengers) {
  string statusMessage = passenger.HasBoarded
    ? "Na pokładzie"
    : CanPassengerBoard(passenger);
  Console.WriteLine($"{passenger.FullName,-23} Group
    {passenger.BoardingGroup}: {statusMessage}");
}
```

Uwielbiam LINQ i jest to dla mnie bezcenne narzędzie do tworzenia prostych i łatwych w utrzymaniu aplikacji, ale trzeba trochę czasu na przyzwyczajenie się do tej składni i notacji funkcji strzałkowych (`=>`).

Muszę też przyznać, że w kodzie LINQ, który przeglądam, spotykam pewne typowe błędy. Przyjrzymy się im w następnym podrozdziale.

Refaktoryzacja instrukcji LINQ

W ostatnim podrozdziale przyjrzymy się paru optymalizacjom, które dość często można zastosować w kodzie LINQ. Opisuję w nim pewne ulepszenia, które są korzystne dla wszystkich baz kodu wykorzystujących tę technologię.

Wybór odpowiedniej metody LINQ

LINQ umożliwia znalezienie wybranego elementu kolekcji na kilka sposobów.

Gdybyśmy mieli obiekt typu `IEnumerable<Passenger>` o nazwie `people` i chcieli znaleźć kogoś według nazwiska, to moglibyśmy napisać kod podobny do poniższego:

LinqExamples.cs

```
PassengerGenerator generator = new();
List<Passenger> people = generator.GeneratePassengers(50);
Passenger me = people.FirstOrDefault(p => p.FullName == "Matt Eland");
Console.WriteLine($"Matt is in group {me.BoardingGroup}");
```

W tym przykładzie użyto metody LINQ `FirstOrDefault`, która przeszukuje kolekcję do znalezienia pierwszej wartości, dla której funkcja strzałkowa zwraca `true`. Funkcja ta znalazłaby pierwszą osobę o imieniu i nazwisku `"Matt Eland"`, zwróciła tę wartość przez metodę `FirsOrDefault` i zapisała ją w zmiennej klasy `Passenger` o nazwie `matt`.

Gdyby natomiast funkcja strzałkowa nie zwróciła `true` dla żadnego elementu, metoda `FirstOrDefault` użyłaby domyślnej wartości dla typu `Passenger`, którą jest `null` w przypadku typów referencyjnych, takich jak klasy.

Wartości domyślne

Na platformie .NET domyślną wartością logiczną jest `false`, typy liczbowe, takie jak `int` i `float`, mają wartość domyślną `0`, a typy referencyjne, na przykład `string`, `List` i inne klasy, domyślnie przyjmują `null`.

Innymi słowy, metoda `FirstOrDefault` znajdzie Matta, jeśli będzie on wśród pasażerów, i zwróci reprezentujący go obiekt lub zwróci `null`, jeśli go nie znajdzie.

Sęk w tym, że następny wiersz próbuje odczytać wartość `matt.BoardingGroup`. To jest w porządku, jeśli uda się znaleźć element. Jeśli jednak nie uda się go znaleźć, to próba dostępu do `BoardingGroup` skończy się wyjątkiem `NullReferenceException`, co prawdopodobnie nie jest zgodne z zamierzeniem programisty.

Sposób rozwiązania tego problemu zależy od naszych oczekiwań.

Kiedy do wyszukiwania elementów w kolekcji używa się technologii LINQ, należy podjąć dwie decyzje:

- Czy nie mam nic przeciwko temu, aby moja funkcja znalazła więcej niż jeden element, czy zależy mi na tym, aby *co najwyżej* dla jednego elementu była zwracana wartość `true`?
- Czy szukany przeze mnie element może w ogóle nie występować w kolekcji?

Pierwsza decyzja dyktuje to, czy użyjemy metody `First` czy `Single`. Metoda `First` znajduje pierwszy element spełniający warunek zapytania i go zwraca. Natomiast metoda `Single` po znalezieniu pierwszego elementu kontynuuje wyszukiwanie, aby sprawdzić, czy pasuje jeszcze jakiś inny element kolekcji. Jeśli tak, następuje zgłoszenie wyjątku

`InvalidOperationException` informującego, że sekwencja zawiera więcej niż jeden pasujący element.

Większość programistów nie lubi oglądać wyjątków po uruchomieniu swojego programu. Czasami jednak trzeba sprawdzić, czy zapytanie znajduje więcej niż jeden element. Ogólnie rzecz biorąc, zawsze lepiej jest natknąć się na błąd wcześniej niż później, w bardziej niejasnych okolicznościach, w których trudno jest stwierdzić, od czego wszystko się zaczęło.

Druga decyzja, jaką należy podjąć przy wyszukiwaniu elementu w kolekcji, dotyczy braku obiektów spełniających warunek zapytania. Jeśli nam to nie przeszkadza, możemy użyć metody `FirstOrDefault` lub `SingleOrDefault` (w zależności od wcześniejszej decyzji dotyczącej tego, czy przeszkadzają nam wielokrotne dopasowania). Jeżeli jednak brak dopasowania jest dla nas nie do przyjęcia, to zamiast metody `FirstOrDefault` lub `SingleOrDefault` użyjemy metody `First` lub `Single`.

Metody `First` i `Single` zgłaszają wyjątek `InvalidOperationException`, gdy sekwencja nie zawiera ani jednego elementu pasującego do zapytania. Jeśli ich użyjemy, a funkcja strzałkowa nie zwróci prawdy dla żadnego z elementów kolekcji, to zostanie zgłoszony wyjątek. To uniemożliwia posługiwanie się wartościami `null`, które mogą pomóc w uproszczeniu kodu.

Wskazówka

Zgłoszenie wyjątku `InvalidOperationException` dokładnie w tym miejscu w kodzie, w którym wystąpił błąd, jest o wiele lepsze niż napotkanie wyjątku `NullReference` ↪`Exception` 30 wierszy dalej i dochodzenie, skąd wzięła się dana wartość.

Inną techniką, za pomocą której można zapobiec pojawieniu się wyjątku `NullReferen` ↪`ceException`, jest **analiza stanu `null`**, której opis znajduje się w rozdziale 10.

Teraz przyjrzymy się sposobom łączenia metod LINQ.

Łączenie metod LINQ

Jedną z zalet LINQ jest możliwość „tworzenia łańcuchów metod" przez wywoływanie kolejnych metod na wynikach zwróconych przez poprzednie. Dzięki temu można na przykład filtrować zbiory elementów za pomocą metody `Where`, zmieniać kolejność wyników za pomocą metody `OrderBy` albo przekształcać wyniki w nowe obiekty za pomocą metody `Select`.

Jednak wraz z rozwojem platformy .NET technologia LINQ wzbogaciła się o kilka bardziej specjalistycznych wersji istniejących metod, przez co tworzenie łańcuchów metod stało się niepotrzebne, a nawet nieefektywne.

Spójrz na przykład na poniższy blok kodu:

```
bool anyBoarded =
  people.Where(p => p.HasBoarded).Any();
```

```
int numBoarded =
  people.Where(p => p.HasBoarded).Count();
Passenger firstBoarded =
  people.Where(p => p.HasBoarded).First();
```

Na pierwszy rzut oka ten kod wygląda dobrze. Każde z tych trzech przypisań do zmiennych filtruje dane, a następnie robi coś z wynikami tego filtrowania. Oczywiście można tu wprowadzić lokalną zmienną dla `people.Where(p => p.HasBoarded)`, ale poza tym ten kod w pierwszej chwili nie budzi zastrzeżeń.

Jednak w LINQ dostępne są przeciążone wersje metod `Any`, `Count`, `First` i paru innych, które przyjmują **predykat** (to tylko wymyślna nazwa funkcji strzałkowej).

Te przeciążone wersje umożliwiają łączenie metod `Where` i innych w bardziej zwięzły sposób:

```
bool anyBoarded = people.Any(p => p.HasBoarded);
int numBoarded = people.Count(p => p.HasBoarded);
Passenger firstBoarded = people.First(p => p.HasBoarded);
```

Ta forma jest nie tylko bardziej zwięzła w zapisie, lecz dodatkowo w niektórych przypadkach może być wydajniejsza.

Kiedyś na przykład, jeśli napisaliśmy instrukcję `people.Where(p => p.HasBoarded).Any()`, to była ona wykonywana od lewej, czyli najpierw program filtrował dużą listę elementów, aby utworzyć z niej mniejszą listę elementów. Potem była wywoływana metoda `Any`, która zwracała wartość `true`, jeśli w nowej liście znalazła przynajmniej jeden pasujący element.

Porównaj to z wersją `people.Any(p => p.HasBoarded)`. Ta metoda przegląda elementy za pomocą pętli i gdy tylko napotka taki, dla którego funkcja strzałkowa zwraca prawdę, może zakończyć pracę, ponieważ już wiadomo, że jej ostatecznym wynikiem będzie prawda.

Zawsze szukaj okazji do użycia tych specjalnych wersji metod LINQ, bo umożliwiają pisanie bardzo zwięzłego i wydajniejszego kodu.

Przekształcanie przy użyciu metody Select

Powiedzmy, że chcemy utworzyć listę łańcuchów dla wszystkich pasażerów, którzy jeszcze nie weszli na pokład samolotu. Dla każdego nazwiska chcemy utworzyć informację w formacie składającym się z nazwiska osoby i grupy pokładowej, do której ta osoba należy. Przykładowy element mógłby wyglądać tak: `"Priya Gupta-7"`.

W tym celu moglibyśmy napisać poniższy kod:

```
List<string> names = new();
foreach (Passenger p in people) {
  if (!p.HasBoarded) {
     names.Add($"{p.FullName}-{p.BoardingGroup}");
  }
}
```

Jednak w LINQ istnieje metoda o nazwie `Select`, która umożliwia przekształcanie formatu elementów, a więc idealnie sprawdzi się w naszej sytuacji.

Wskazówka

Informacja dla osób znających JavaScript: metoda `Select` jest podobna do funkcji `Map`.

Wersja powyższego kodu utworzona z użyciem metody `Select` wygląda następująco:

```
List<string> names =
        people.Where(p => !p.HasBoarded)
                .Select(p => $"{p.FullName}-{p.BoardingGroup}")
                .ToList();
```

Tutaj metoda `Where` wybrała z kolekcji wszystkich pasażerów, którzy jeszcze nie weszli na pokład, a następnie metoda `Select` przekształciła te obiekty na łańcuchy.

Metoda `Select` działa nie tylko z łańcuchami. Można jej używać z dowolnym typem danych, w tym z liczbami całkowitymi, innymi obiektami, listami, a nawet z **typami anonimowymi** i **krotkami**.

Krótko mówiąc, jeśli masz kolekcję obiektów w określonym kształcie i chcesz je przekształcić do innej postaci, to warto rozważyć użycie do tego celu metody `Select`.

Przegląd i testowanie kodu po refaktoryzacji

Choć w tym rozdziale nie zmodyfikowaliśmy dużej ilości kodu, to fragmenty, którymi się zajęliśmy, stały się krótsze oraz czytelniejsze, bardziej zrozumiałe i łatwiejsze do modyfikacji.

Właśnie dlatego przeprowadzamy refaktoryzację. Działania te powinny ułatwiać utrzymanie kodu aplikacji i spłacać strategiczne raty długu technicznego, które grożą pojawieniem się błędów i opóźnień w przyszłości.

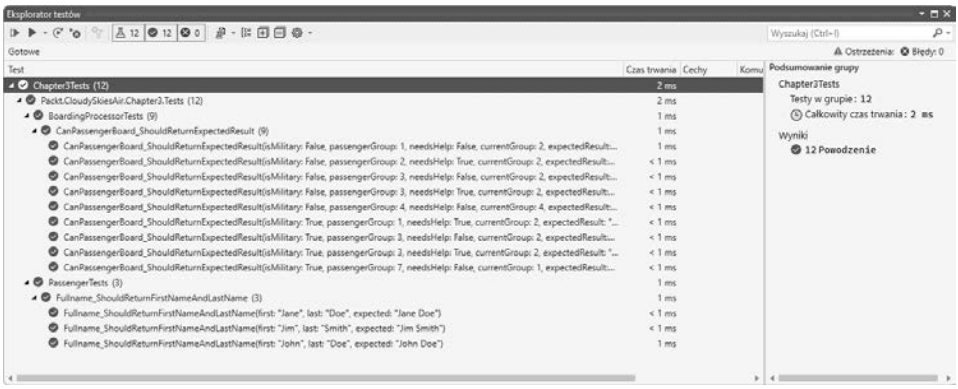
Kod po refaktoryzacji

Ostateczna wersja zrefaktoryzowanego kodu z tego rozdziału znajduje się pod adresem *https://ftp.helion.pl/przyklady/refawc.zip* w folderze o nazwie *Chapter03/Ch3 ↪RefactoredCode*.

Jako że sztuka refaktoryzacji polega na modyfikowaniu kodu tak, aby nie zmienić jego funkcjonalności, zanim przejdziemy dalej, musimy przetestować naszą aplikację.

Więcej na temat ręcznych i automatycznych testów piszę w rozdziale 6. Na razie tylko uruchom testy kliknięciem opcji *Uruchom wszystkie testy* w menu *Test* środowiska Visual Studio.

Spowoduje to pojawienie się okna *Eksplorator testów* pełnego zielonych symboli, jak widać na rysunku 3.10.



Rysunek 3.10. Zaliczone testy programu z tego rozdziału

Czas na podsumowanie zdobytych wiadomości.

Podsumowanie

W tym rozdziale opisałem techniki refaktoryzacji usprawniające kontrolę przepływu sterowania, tworzenie obiektów, iterowanie przez kolekcje i pisanie wydajniejszego kodu przy użyciu LINQ.

Każda z opisanych technik refaktoryzacji jest kolejnym narzędziem, dzięki któremu można poprawić czytelność i ułatwić utrzymanie kodu w określonych warunkach. W miarę zdobywania doświadczenia w refaktoryzacji będziesz coraz lepiej rozpoznawać, kiedy warto stosować określone techniki, aby ulepszyć dany kod.

W następnym rozdziale nie będziemy pracować nad pojedynczymi wierszami kodu, tylko przyjmiemy odrobinę szerszą perspektywę — będziemy refaktoryzować całe metody kodu C#.

Pytania

Odpowiedz na poniższe pytania, aby sprawdzić, co udało Ci się zapamiętać z tego rozdziału:

1. Co jest ważniejsze: zwięzły kod czy czytelny kod?
2. Przejrzyj plik z kodem w projekcie, nad którym obecnie pracujesz. Co możesz powiedzieć o znajdujących się w nim instrukcjach `if`?
3. Czy zawiera on dużo zagnieżdżonych instrukcji `if`?
4. Czy w warunkach Twoich instrukcji `if` znajdują się fragmenty powtarzalnej logiki?
5. Czy widzisz jakiekolwiek okazje do poprawienia kodu przez odwrócenie instrukcji `if` lub zamianę ich na instrukcję albo wyrażenie `switch`?
6. Czy według Ciebie Twój zespół w pełni wykorzystuje możliwości technologii LINQ podczas pracy z kolekcjami? Jakie możliwości wprowadzenia ulepszeń widzisz?

Dalsza lektura

Na poniższych stronach znajdziesz dodatkowe informacje na temat omówiony w tym rozdziale:

- Wyrażenia `switch`: *https://learn.microsoft.com/en-US/dotnet/csharp/language-reference/operators/switch-expression*
- Różnice między interfejsami kolekcji .NET: *https://newdevsguide.com/2022/10/09/understanding-dotnet-collection-interfaces/*
- Składnia zapytań i składnia metod w LINQ: *https://learn.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/query-syntax-and-method-syntax-in-linq*
- Zakresy danych: *https://learn.microsoft.com/en-us/dotnet/csharp/tutorials/ranges-indexes*
- Funkcje strzałkowe i operator lambda: *https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-operator*

Dług techniczny?
Poznaj najlepsze narzędzia do refaktoryzacji!

Termin *dług techniczny* oznacza zbiór skrótów, niedociągnięć i potworków projektowych, które powstają w trakcie ewolucyjnego rozwoju programu. Jeśli się ich nie zlikwiduje, mogą dramatycznie spowolnić pracę nad programem. Rozwiązywanie tego typu problemów jest nazywane refaktoryzacją i każdy programista powinien umieć ją przeprowadzać.

Dzięki tej przystępnej, świetnie napisanej książce dowiesz się, czym jest dług techniczny, co prowadzi do jego powstawania i w jaki sposób można go bezpiecznie zrefaktoryzować przy użyciu nowoczesnych narzędzi dostępnych w środowisku Visual Studio, a także najnowszych składników języka C# 12 i platformy .NET 8. Nauczysz się też korzystać z zaawansowanych testów jednostkowych tworzonych przy użyciu xUnit i takich bibliotek jak Moq, Snapper czy Scientist.NET. Dowiesz się, jak stosować zasady SOLID, aby tworzyć łatwy w utrzymaniu kod, poznasz również techniki programowania defensywnego, których można używać w nowszych wersjach C#. Ponadto nauczysz się przeprowadzać analizy kodu i pisać własne analizatory Roslyn do wykrywania i rozwiązywania problemów typowych dla Twojego projektu.

Najciekawsze zagadnienia:

- najważniejsze informacje o długu technicznym
- różne sposoby refaktoryzacji klas, metod i wierszy kodu
- efektywne testy jednostkowe
- zasady SOLID i tworzenie kodu łatwego w utrzymaniu
- korzystanie ze sztucznej inteligencji GitHub Copilot
- standardy kodowania w zwinnych zespołach

Matt Eland otrzymał od Microsoftu tytuł MVP w dziedzinie sztucznej inteligencji. Z platformy .NET korzysta od 2001 roku. Obecnie jest specjalistą w zakresie sztucznej inteligencji i starszym konsultantem w Leading EDJE. Działa również jako prelegent i współorganizator Central Ohio .NET Developers Group.

Helion
helion.pl
HELION S.A.
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-1680-7
9 788328 916807
Cena: 99,00 zł
packt