POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania
Inżynieria Oprogramowania i Baz Danych

Autor: **Filip Kwiatkowski**

Nr albumu: **s17137**

Programowanie generyczne we współczesnych językach programowania ze szczególnym uwzględnieniem Konceptów i Zakresów z C++20

Praca magisterska

Promotor:
Dr inż. Mariusz Trzaska

Warszawa, lipiec 2023

Streszczenie

Praca przedstawia i porównuje współczesne języki programowania pod kątem udostępnianych przez nie narzędzi do programowania generycznego. Wprowadza ideę, problematykę i wagę tych elementów językowych oraz pokazuje, w jaki sposób mogą one ewoluować, zwykle bazując na nowych potrzebach biznesowych oraz na obserwacji ewolucji konkurencyjnych narzędzi. Praca koncentruje się głównie na porównywaniu podejść i dostępnych narzędzi w językach Java, Python oraz C++ ze szczególnym uwzględnieniem tego ostatniego. Owe szczególne uwzględnienie dotyczy dokładnego omówienia historii języka w kontekście generycznych narzędzi wraz z najnowszymi elementami, tj. Konceptami i Zakresami. Dodatkowo, w ramach pracy powstało rozszerzenie fragmentu biblioteki standardowej języka C++, który obrazuje szereg problemów i ważnych elementów, na które trzeba zwrócić uwagę tworząc podobne narzędzia. Na koniec praca krótko omawia alternatywne rozwiązania oraz przykładowy proces testowania stworzonego rozwiązania.

Słowa kluczowe

Programowanie generyczne, szablony, typy generyczne, C++, Koncepty, Concepts, Zakresy, Ranges, biblioteka standardowa, STL.

Podziękowania

Autor pracy pragnie podziękować wszystkim osobom, które okazały szczególne wsparcie w tworzeniu treści zawartych w tym dokumencie i przyczyniły się do zwiększenia jakości samej pracy, w tym: promotorowi doktorowi Mariuszowi Trzasce – za ogromną pomoc zarówno merytoryczną, w postaci znacznej liczby sugestii dotyczącej jakości i treści, jak i motywacyjną, w postaci zachęcenia do stworzenia pracy o obszerniejszej i ciekawszej treści, niż pierwotnie zakładano; społeczności Zaprogramuj Życie – za dostarczenie opinii dotyczących wybranych przykładów oraz personalnych sugestii dotyczących kluczowych elementów, które urozmaiciły treść pracy; oraz bliskim, którzy okazywali nieustanne wsparcie wszelkiego rodzaju w trakcie pisania całej pracy.

Spis Treści

1 WSTĘP ... 6
1.1 Cel pracy ... 6
1.2 Konwencje przyjęte w pracy ... 7
1.2.1 Argument a parametr ... 7
1.2.2 Metoda a funkcja ... 7
1.2.3 Listingi kodu ... 7
1.2.4 Kolekcja a kontener ... 9
1.2.5 Styl kodu ... 9
1.2.6 Moduł nowostworzonego narzędzia ... 9
1.2.7 Pozostałe nazewnictwo ... 9
1.3 Rezultaty pracy ... 10
1.4 Organizacja pracy ... 10
2 ZASTOSOWANIA, ASPEKTY I PROBLEMY PROGRAMOWANIA GENERYCZNEGO ... 12
2.1 Algorytmy i struktury danych ... 13
2.2 Typy danych ... 14
2.3 Problem duplikacji kodu ... 15
2.4 Trudności w wytwarzaniu generycznego oprogramowania ... 16
2.4.1 Przykład programowania generycznego w języku Python ... 17
2.4.2 Przykład programowania generycznego w języku Java ... 18
2.4.3 Przykład programowania generycznego w języku C++ ... 20
3 HISTORIA JĘZYKA C++ W KONTEKŚCIE OGÓLNIE POJĘTEGO PROGRAMOWANIA GENERYCZNEGO ... 22
3.1 C++ przed pierwszym oficjalnym standardem ... 23
3.1.1 Wprowadzenie szablonów ... 25
3.2 Pierwszy standard i jego poprawki ... 26
3.3 Nowoczesny C++ ... 26
3.3.1 Zmiany praktyk programistycznych w ramach nowoczesnego C++ ... 28
3.4 Kompletny C++ ... 34
3.4.1 Błędy zgłaszane przez narzędzia przy używaniu narzędzi generycznych ... 35
3.4.2 Koncepty ... 39
3.4.2.1 Tworzenie zestawu generycznych narzędzi, które akceptują grupę typów ... 39
3.4.2.2 Tworzenie własnych Konceptów i ograniczeń dla typów generycznych ... 45
3.4.2.3 Reprezentowanie ograniczeń typów generycznych w celu zwiększenia klarowności interfejsu ... 48
4 WPŁYW NOWYCH ELEMENTÓW JĘZYKA C++ DOTYCZĄCYCH PROGRAMOWANIA GENERYCZNEGO NA ROZWÓJ INNYCH NARZĘDZI ... 53
4.1 Zakresy ... 53
4.1.1 Algorytmy zakresowe ... 56
4.1.2 Projekcje ... 58
4.1.3 Adaptery zakresów ... 65
4.1.3.1 Paradygmat funkcyjny ... 66
4.1.3.2 Widoki ... 68
5 POMNIEJSZE USPRAWNIENIA NARZĘDZI JĘZYKA C++ ... 74
5.1 Uproszczenie wybranych konstrukcji programistycznych ... 74
5.2 Usprawnienia środowisk programistycznych ... 75
6 IMPLEMENTACJA ROZSZERZENIA KONTENERÓW BIBLIOTEKI STANDARDOWEJ C++ ... 78
6.1 Niekompletna natura biblioteki standardowej języka C++ ... 78
6.2 Identyfikacja elementów kolekcji biblioteki standardowej, które można wzbogacić o uzakresowione konstrukcje ... 79
6.3 Implementacja elementów kolekcji biblioteki standardowej, które można wzbogacić o uzakresowione konstrukcje ... 82

6.3.1 Rozwiązanie problemu braku wirtualnego destruktora w typach bazowych za pomocą prywatnego dziedziczenia ... 83
6.3.2 Problem widoczności odziedziczonych elementów w kontekście dziedziczenia prywatnego ... 83
6.3.3 Definicja szablonów metod akceptujących zakres wartości w postaci pojedynczego obiektu ... 86
6.3.4 Zakresy z elementami będącymi r-wartościami ... 90
6.3.5 Problem z operatorami porównującymi ... 91
6.3.5.1 Podejścia różnych języków do rezultatu porównywania ... 92
6.3.5.2 Implementacja operatorów porównujących dla kontenerów z masters ... 93
6.3.6 Implementacja wskazówek dedukcji argumentów szablonowych dla klas (CTAD) ... 95
6.3.7 Pozostałe kontenery z masters ... 99

7 TESTOWANIE STWORZONEGO NARZĘDZIA ... 100

8 ALTERNATYWNE ROZWIĄZANIA ... 101

8.1 Implementacja MSVC ... 101
8.2 Implementacje pozostałych wiodących dystrybutorów kompilatorów i bibliotek standardowych ... 101

9 PODSUMOWANIE I WNIOSKI ... 102

PRACE CYTOWANE ... 104

1 Wstęp

Współczesne oprogramowanie, a właściwie kod składający się na nie, można podzielić na wiele sposobów. Jednym z tych podziałów jest odróżnienie kodu biznesowego od kodu generycznego. Kodem biznesowym określa się fragmenty, które bezpośrednio realizują unikalne wymagania biznesowe tworzonej aplikacji. Kodem generycznym jest wtedy logika, której poprawność i użyteczność nie zależy od dziedziny, w której jest wykorzystywana.

Oprogramowanie to połączenie jednego i drugiego. Potrzeba istnienia kodu biznesowego jest oczywista – to on odpowiada za realizowanie celu, dla którego powstaje tworzona aplikacja. Jednakże wytwarzanie od zera wszystkich narzędzi wymaganych do sprawnego tworzenia logiki biznesowej byłoby wysoce nieefektywnym procesem. Dlatego z perspektywy programistów bardzo istotnym jest dysponowanie dobrej jakości kodem generycznym, który ze względu na swoją uniwersalność może być użyty w procesie tworzenia przeróżnych modułów, które reprezentują biznesową logikę.

1.1 Cel pracy

Niniejsza praca koncentruje się na porównaniu i analizie narzędzi programowania generycznego we współczesnych językach programowania. W kontekście tych porównań i analizy została przedstawiona szczegółowa historia ewolucji języka C++ ze szczególnym uwzględnieniem elementów, które dotyczą tematyki tytułu pracy.

Istotnym jest zrozumienie różnych aspektów, które mają wpływ na jakość wytwarzania narzędzi generycznych we współczesnych językach. Praca ma na celu zobrazować nie tylko trudność używania narzędzi generycznych, ale też ich tworzenia oraz rozszerzania.

Trudność użytkowania nie ogranicza się do spójności i konsekwentności interfejsów, ale również i do (czasem wątpliwej) jakości błędów zgłaszanych przez narzędzia takie jak kompilator, linker czy interpreter. Praca ma na celu wskazać różnice między jakością raportowania wspomnianych błędów w różnych językach programowania.

W celu porównania różnych podejść do programowania generycznego zostały wybrane dwa główne języki poza C++ – Java oraz Python. W niektórych miejscach pojawiają się wzmianki o innych językach, lecz główne porównania i analizy ograniczają się do wymienionej trójki. Taka decyzja jest argumentowana relacją tych języków. Wszystkie trzy są wśród najpopularniejszych języków programowania na świecie według praktycznie wszystkich ankiet i badań od wielu lat (np. [1]), a co więcej dzielą one częściowo wspólną przeszłość w procesie swojej ewolucji. Praca ma zatem na celu również przybliżyć obecny stan tych narzędzi, a nie tylko przedstawiać kontekst historyczny.

Dodatkowo, w ramach pracy powstał relatywnie obszerny fragment generycznej biblioteki programistycznej. Praca przybliża proces tworzenia takiego narzędzia i obrazuje problemy, z którymi można się spotkać próbując takowe stworzyć.

1.2 Konwencje przyjęte w pracy

Z uwagi na charakter pracy, który podchodzi bardzo teoretycznie do analizy narzędzi, które są czysto praktyczne, należy dokładnie określić konwencje, które zostały przyjęte w pracy, aby uniknąć niejednoznaczności, niekonsekwentności i niespójności.

1.2.1 Argument a parametr

Praca wprowadza rozróżnienie na termin *argument* oraz termin *parametr*. Mówiąc krótko, parametry dotyczą zmiennych widniejących w deklaracji / definicji, a argumenty to realne dane, które są wysyłane do używanego narzędzia (np. funkcji).

1.2.2 Metoda a funkcja

Wiele języków korzysta z różnych, czasem między sobą sprzecznych, definicji terminów *metoda* oraz *funkcja*. W ramach niniejszej pracy te terminy są używane następująco:

- *Funkcja* oraz *wolna funkcja* mają to samo znaczenie. Reprezentują one wywoływalny fragment kodu, który nie jest częścią żadnej klasy.
- *Metoda* to funkcja, która jest częścią klasy.

Powyższe rozróżnienia stawia kilka jasnych implikacji:

- Język Java, zgodnie z przedstawioną konwencją, w ogóle nie wspiera istnienia funkcji, bo każda funkcja musi być częścią klasy. Oznacza to, że taka funkcja jest metodą.
- Język Python oraz C++ wspierają tworzenie zarówno funkcji jak i metod. W języku C++ formalne rozróżnienie mówi o nieco innych terminach: *wolna funkcja* (ang. *free function*) oraz *funkcja składowa* (ang. *member function*). Dla klarowności i uproszczenia terminologii te dwa terminy zostały skrócone do *funkcji* i *metody*.

Dodatkowo, gdy tekst pracy referuje do nazwy funkcji lub metody, zawsze po jej nazwie są stawiane puste nawiasy, nawet jeżeli przyjmuje ona jakieś parametry. Decyzja ta jest argumentowana czytelnością dotyczącą prostego rozróżnienia typów oraz zmiennych od funkcji i metod.

W ramach tego samego kontekstu, o którym mowa w poprzednim akapicie, nazwy metod są poprzedzane kropką aby dodatkowo ułatwić ich odróżnienie od funkcji.

1.2.3 Listingi kodu

Listingi kodu zawierają ponumerowane linie. Jeżeli następujące po sobie linie nie są opatrzone kolejnymi liczbami naturalnymi (zamiast tego jest pusta przestrzeń), to oznacza to, że ostatnia linia opatrzona numerem została sformatowana tak, że w niniejszej pracy zajmuje ona więcej niż jeden wiersz.

Podsumowując, jeżeli jakaś linia jest opatrzona numerem, a następna (lub następne) nie, to reprezentuje to jedną logiczną linijkę kodu sformatowaną na kilka linii w tekście pracy.

W przypadku pominięcia fragmentu kodu stosowany jest znak wielokropka (…). Należy zwrócić szczególną uwagę na fakt, że jest to zupełnie coś innego niż trzy kropki (`...`). W kontekście języka C++, trzy kropki zwykle reprezentują paczkę parametrów szablonowych. W języku Java jest to zapis arbitralnej liczby argumentów, które są zbierane do pojedynczego parametru tablicowego. Język Python traktuje ten operator jako zamiennik automatycznie dostosowującej się liczby wymiarów indeksujących.

Przykład można zaobserwować w Listing 1.

Listing 1: Przykład sformatowania pojedynczej linii kodu na kilka wierszy w tekście pracy.

```
1  auto function(
2          std::filesystem::recursive_directory_iterator
   veeeeeeeeeeeeeeeeeryLongIteratorName
3  ) -> void { … }
```

Niektóre fragmenty kodu zawierają wielolinijkowe byty, które mogą być rozbite na wiele linii bez zmiany semantycznego znaczenia. Są to, między innymi:

- Wywołania funkcji i metod, których listy argumentów są rozbite na wiele linijek;
- Deklarowanie szablonów lub generycznych bytów, gdzie deklaracja parametrów szablonowych / generycznych może znajdować się na tej samej linii lub linijkę wcześniej / później.

W takich przypadkach czasami te fragmenty kodu są zachowywane w tej samej linii, a czasem prezentowane jako kolejne, osobne linie. Różnice polegają na chęci jak największego zoptymalizowania listingów pod względem czytelności. Przykładowo, Listing 2 przedstawia alternatywne deklaracje szablonów struktur. W jednej z nich słowo kluczowe `template` zostało umieszczone w tej samej linii co słowo kluczowe `struct` z uwagi na to, że pełna linijka kodu jest dosyć krótka. W drugiej deklaracji słowa te zostały umieszczone w osobnych linijkach, aby zwiększyć czytelność i przejrzystość kodu.

Listing 2: Przykład konwencji sprecyzowania parametrów szablonowych w tej samej oraz w osobnej linii.

```
1  template <typename T> struct Example { };
2
3  template <typename T>
4  struct ExampleWithFields {
5      T first;
6      T second;
7  };
```

W związku z tym, że praca koncentruje się głównie na listingach zawierających kod w języku C++, warto zdefiniować konwencję importowania nazw i bibliotek do zamieszczonych snippetów kodu. O

ile nie jest to dokładnie omówione, to listingi z innych języków są pozbawione fragmentów odpowiadających za dołączanie nazw / bibliotek, np. za pomocą słowa kluczowego `import` w Javie czy Pythonie.

W przypadku listingów C++owych, instrukcje `#include` są zawarte tylko wtedy, gdy mają istotne znaczenie dla przedstawianej logiki i dołączają narzędzie, które nie było wcześniej omawiane lub natura tego narzędzia jest nietrywialna.

1.2.4 Kolekcja a kontener

Podobnie jak w przypadku terminów metoda i funkcja objaśnionych w rozdziale **1.2.2**, terminy *kolekcja* i *kontener* różnią się zależnie od kontekstu i języka. Niektórzy wprowadzają następujące rozróżnienie:

- *Kolekcja* to obiekt reprezentujący zbiór elementów, którego celem jest oferowanie operacji na tym zbiorze takich jak: dodawanie, usuwanie, modyfikowanie i zmienianie ich kolejności.
- *Kontener* to obiekt opakowujący jakiś inny element lub ich zbiór, ale nieudostępniający semantyki modyfikowania tego zbioru, w szczególności ich liczności. Przykładem byłyby takie typy jak: `std::move_iterator` z C++, `Integer` z Javy, `StringBuilder` z Javy (jako opakowanie na tablicę znaków z dodatkowym interfejsem) czy `Tuple` z Pythona.

W niniejszej pracy terminy te są używane wymiennie. To dlatego, że w ramach dwóch bardzo istotnych omawianych języków – Javy oraz C++ – mają one kompletnie odwrotne znaczenie. W Javie termin kolekcja odnosi się do tego, co w C++ jest reprezentowane przez termin kontener.

1.2.5 Styl kodu

W listingach niniejszej pracy został zastosowany styl konsekwentny z tym z Javy (mieszanka **PascalCase** oraz **camelCase**). To dlatego, że mimo iż większość listingów kodu dotyczy języka C++, nie posiada on ustandaryzowanej konwencji dotyczącej nazw czy jakiegokolwiek innego formatowania kodu. Mimo tego, elementy rozszerzające bibliotekę standardową tego języka będą spójnie korzystać z notacji **snake_case**.

1.2.6 Moduł nowostworzonego narzędzia

Rozszerzenie biblioteki standardowej języka C++, które powstało jako skutek uboczny prac badawczych wykonanych w ramach niniejszej pracy zostało zamknięte w dedykowanej przestrzeni nazw o nazwie `masters`. Nazwa ta jest nawiązaniem do angielskiego terminu oznaczającego stopień magistra – *master's degree*.

1.2.7 Pozostałe nazewnictwo

W tekście pracy można znaleźć termin *legalny kod* pod różnymi postaciami i odmianami. Referuje on do fragmentu kodu (lub bardziej ogólnie – do zestawu operacji), które w ramach jakiegoś sprecyzowanego kontekstu mogą zostać użyte bez bycia przyczyną błędów budowania aplikacji.

1.3 Rezultaty pracy

Treść zawarta w niniejszej pracy stanowi podwaliny zrozumienia procesów, które ukształtowały ewolucję języka C++ w kontekście programowania generycznego. Obrazuje ona również wagę świadomości istnienia alternatywnych narzędzi wraz z ich adaptacją, która może się okazać kluczowa przy tworzeniu nowych narzędzi generycznych (i nie tylko). Jawnie zaznacza mocne i słabe strony różnych rozwiązań wspierających programowanie generyczne we współczesnych językach programowania.

Stworzona implementacja rozszerzająca bibliotekę standardową C++ może być podstawą do przykładowego procesu tworzenia weryfikacji koncepcji (ang. *proof of concept*) w ramach rozszerzania narzędzi generycznych.

Wspomniana implementacja powstawała jako efekt uboczny pracy. Obrazuje ona, jak nowe narzędzia programowania generycznego (niektóre z nich mające swoje odpowiedniki w innych językach) zwiększają jakość produkowania narzędzi szerokiego zastosowania. W związku z tym praca koncentruje się również na zaznaczeniu wagi ewolucji języka, co pozwala zobrazować istotność analizy przeszłości i alternatywnych narzędzi w celu osiągnięcia jak najwydajniej funkcjonującego ekosystemu narzędzi, na który składają się takie elementy jak:

- narzędzia językowe;
- narzędzia bibliotek standardowych;
- narzędzia bibliotek zewnętrznych;
- oprogramowanie wspierające proces wytwarzania kodu, takie jak:
    - środowisko programistyczne (IDE);
    - kompilatory;
    - linkery;
    - statyczne analizatory.

1.4 Organizacja pracy

Praca została podzielona na 9 rozdziałów, z czego faktyczna treść dotycząca badanej tematyki zaczyna się od rozdziału drugiego.

Drugi rozdział traktuje o przybliżeniu idei, wagi i potencjalnych trudności wynikających ze stosowania programowania generycznego. Treść rozdziału opiera się o praktyczne przykłady krótko omawiające generyczne algorytmy i struktury danych wraz z ich uproszczonymi implementacjami w wybranych językach programowania. Rozdział ten tworzy podwaliny do dalszych rozważań w kontekście opisów bardziej konkretnych przykładów omówionych w dalszej części pracy.

Trzeci rozdział omawia historię języka C++ ze szczególnym uwzględnieniem ewolucji jego narzędzi do programowania generycznego. Ukazuje przykłady użycia przytoczonych narzędzi w kontekście

przedstawionych w poprzednim rozdziale problemów, z zachowaniem chronologicznej kolejności ich dodawania do języka, wraz z samą motywacją dodania ich do języka.

Rozdział czwarty dokładnie omawia bibliotekę Zakresów, która jest jednym z najnowszych narzędzi dodanych do języka C++. Porównuje ją z alternatywnymi podejściami osiągającymi podobne cele narzędziami z innych języków. Ma na celu, między innymi, pokazać możliwości, które narzędzia programowania generycznego udostępniają.

Rozdział piąty jest zwieńczeniem dwóch poprzednich rozdziałów. Wskazuje dodatkowe benefity płynące z ewolucji narzędzi programowania generycznego – również w kontekście narzędzi wspierających pracę programisty.

Rozdział szósty został poświęcony szczegółowemu omówieniu implementacji rozszerzenia biblioteki standardowej języka C++ o nowe zachowania szablonów kolekcji.

Rozdział siódmy krótko omawia sposoby na testowanie narzędzia stworzonego według przedstawionych wcześniej założeń i koncepcji.

Rozdział ósmy wprowadza obecny stan ewolucji biblioteki języka w kontekście nowostworzonego narzędzia i obrazuje różny stopień wspierania nowych elementów języka C++ przez różne kompilatory.

Rozdział dziewiąty krótko przedstawia podsumowanie i wnioski.

W ramach każdego rozdziału został nałożony ogromny nacisk na dostarczenie listingów kodu obrazujących przedstawiany problem. Wszystkie listingi kodu zostały stworzone dzięki natywnemu wsparciu narzędzia MS Word do kopiowania bogatych treści. Dzięki temu kod stworzony i przetestowany w wybranym środowisku programistycznym (głownie IDE od firmy JetBrains oraz Microsoft) mógł być trywialnie skopiowany i wklejony do tworzonego dokumentu z zachowaniem pełnego formatowania.

2 Zastosowania, aspekty i problemy programowania generycznego

Oczywistym jest, że wiele problemów, z którymi spotykają się programiści, jest do siebie podobnych. Idea projektowania jednego rozwiązania, które się aplikuje do wielu sytuacji jest znana i praktykowana od bardzo dawna.

Istnieje pewne piękno w stanie rzeczy, który pozwala na to, aby dany algorytm był generyczny i, zależnie od pewnych, zdefiniowanych punktów dostosowania (ang. *customization points*), mógł być aplikowany do praktycznie każdego wariantu danego problemu.

Dobrym przykładem jest algorytm sortowania. Świat algorytmiki, tak obszerny i ciekawy, jest również miejscem narodzin narzędzi, bez których świat programistyczny byłby zdecydowanie mniej elegancki. Zachwycające jest to, że akceptowalnym przybliżeniem może być stwierdzenie, że dany algorytm sortowania jest w stanie posortować dowolny rodzaj danych, *o ile jest w stanie te dane porównywać i zamieniać miejscami*.

Takie przybliżenie już wystarcza, aby zacząć formalizować wymagania dotyczące danych, które można sortować. Warto zauważyć, że współczesne języki programowania oferują szeroki wachlarz narzędzi, które pozwalają dobrze zdefiniować potrzebne do spełnienia cechy danych, jeżeli chce się je wykorzystać w danych algorytmie.

Przykładowo, sygnatury algorytmów sortujących listy w języku Java (wersja 19) wyglądają następująco (Listing 3):

Listing 3: Sygnatury przeciążeń metody `sort()` dla list w języku Java 19.

```
1  public static <T extends Comparable<? super T>>
2  void sort(List<T> list) { … }
3
4  public static <T>
5  void sort(List<T> list, Comparator<? super T> c) { … }
```

Nie trzeba na ten moment dokładnie rozumieć działania mechaniki typów generycznych, które są wykorzystywane w Javie – będą one dokładnie omówione w dalszej części pracy.

Należy skoncentrować się na sprecyzowanych wymaganiach dotyczących typów, które *można sortować* tym algorytmem. Pierwsze przeciążenie metody `sort()` pozwala na posortowanie dowolnej listy, o ile jej elementy mają zdefiniowaną jakąś ustandaryzowaną logikę sortowania (ograniczenie `T extends Comparable<? Super T>>`).

Czasami jednak pojawia się sytuacja, gdzie nie istnieje jakiś domyślny sposób porównywania danych lub ten domyślny sposób jest w danym miejscu nieodpowiedni. W takich sytuacjach należy skorzystać z drugiego przeciążenia metody `sort()`, która może posortować dowolny typ danych, o ile programista dostarczy dodatkowy mechanizm, który pozwoli na jego porównywanie (obiekt `c` pozwalający na porównanie elementów z przekazanej listy).

Jest to przykład kodu o wysokiej jakości architekturze. Została wykonana dodatkowa, skomplikowana praca, aby zdefiniować formalne wymagania w postaci interfejsów zapewniających dodatkowe bezpieczeństwo oraz wygodę użytkowania tego algorytmu nawet w niestandardowych przypadkach. Powstałe narzędzie jest elastyczne, a co za tym idzie – łatwe w zmianie lub rozszerzaniu.

Efektywnie, zestaw zaprezentowanych przeciążeń pozwala na posortowanie dowolnego typu danych, o ile dostarczy się programowi odpowiednią logikę służącą do porównywania elementów.

2.1 Algorytmy i struktury danych

Sortowanie, a właściwie szeroko pojęte algorytmy, nie są jednak jedynymi dziedzinami, gdzie można znaleźć zastosowanie dla programowania generycznego. Bardzo istotnym elementem wytwarzania współczesnego oprogramowania jest połączenie algorytmów ze strukturami danych.

Już od dawna wiadomo, że czasami równie istotnym jest sposób przechowania danych jak i sama ich natura. Oczywiście rodzaj samych danych kształtuje intencję aplikacji, ale odpowiednio dobrane struktury i kolekcje kształtują sposób ich wykorzystania. Bezpośrednio wpływają na jakość oprogramowania poprzez – docelowo – ułatwianie manipulacji danymi biznesowymi.

Tak samo jak generyczny algorytm, generyczna struktura danych powinna, z założenia, wspierać ujednoliconą i ustandaryzowaną reprezentację danych. Przez *reprezentację danych* należy rozumieć:

- Przechowywanie danych, np. rozkład pamięci (ciągły (ang. *contiguous*), lub nieliniowy (ang. *non-contiguous*);
- Sposób dostępu do danych, np. poprzez indeksy, iteratory, klucze, itp;
- Modyfikowalność danych, czyli czy (i jeżeli tak, to które) dane można modyfikować oraz czy można dodawać lub usuwać elementy z danej struktury.

Trafnym byłoby podjąć próbę naszkicowania korelacji między tymi charakterystykami struktur danych oraz wspomnianych wcześniej algorytmów – generyczne algorytmy muszą, a raczej powinny, również wspierać pewien ustandaryzowany sposób działania.

Celowo zostało tutaj użyte stwierdzenie „*muszą, a raczej powinny*". Naturalnie ta cecha nie jest wymagana do zaprojektowania działającej biblioteki oferującej dane operacje. Mimo wszystko konsekwentność, spójność i jednolitość (wszystkie - ang. *consistency*) narzędzi jest jednym z kluczy do sukcesu ich popularności i przyjemności płynącej z ich użytkowania.

Scott Meyers, jedna z najbardziej rozpoznawalnych i szanowanych postaci w kręgu użytkowników języka C++, twierdzi [2], że powinno się projektować interfejsy (API) zgodnie z Zasadą Najmniejszego Zdziwienia (ang. *Principle of Least Astonishment*).

Biorąc pod uwagę fakt, że współczesne programy działają na ogromnej liczbie różnych typów danych, które z kolei z założenia nie powinny być na tyle „słabe", że można je do woli podmieniać, pojawia się problem reprezentowania i pracy na różnych typach. Należy tworzyć rozwiązania uniwersalne, a próby

powtórzenia implementacji dla różnych typów danych będą skutkować ogromną duplikacją kodu. Programowanie generyczne dąży do rozwiązania tego problemu.

2.2 Typy danych

Ogromna siła i potencjał w obecnie wytwarzanym oprogramowaniu znajduje swoje źródło w narzędziach, które pomagają w pracy programisty. Mowa tutaj o kompilatorach, linkerach, środowiskach programistycznych (ang. *IDE*) czy narzędziach do usuwania specyficznych błędów, takich jak błędy adresowe czy błędy przy używaniu wątków, np. *Address Sanitizer* czy *Thread Sanitizer*.

Narzędzia te korzystają z metadanych programu w celu jego jak najdokładniejszej analizy, dzięki czemu mogą one lepiej modelować strukturę aplikacji w celu detekcji błędów. Jednym z rodzajów takich metadanych są typy danych tworzonych obiektów[1] w programie. Ich przydatność jest bezdyskusyjna – uznaje się, że przede wszystkim pomagają one w detekcji błędów, ale dodatkowo służą również w utrzymaniu spójności modułów aplikacji [3].

Typy danych muszą być od siebie odseparowane, tj. gdy program oczekuje dostarczenia obiektu o danym typie, to nie powinien akceptować próby dostarczenia danych innego typu. Oczywiście istnieją wyjątki, np. w postaci polimorficznych hierarchii realizowanych przez klasy czy interfejsy. Nie są one jednak pogwałceniem zasady separacji typów. Hierarchie polimorficzne są z reguły relatywnie płytkie. Typy będące częścią tej hierarchii dotyczą podobnych problemów powiązanych z daną dziedziną biznesową.

Polimorfizm można zatem nazwać pracą na wielu typach w kontekście kodu biznesowego. Natomiast problemy opisywane wyżej dotyczą innego, jeżeli nie stricte szerszego zakresu – pracy na wielu typach danych w kontekście kodu uniwersalnego, czyli generycznego.

Łatwo można zobrazować problem dotyczący przechowywania lub pracy nad danymi, które nie mają ze sobą nic wspólnego. Tak samo poprawna byłaby potrzeba przechowywania danych dotyczących klientów z zachowaną kolejnością, którą potem się odwraca, jak i potrzeba przechowywania danych liczbowych dotyczących pewnej symulacji komputerowej z zachowaną ich kolejnością, którą potem się odwraca.

W powyższym przykładzie mamy do czynienia z dwoma, osobnymi biznesowymi przypadkami (przechowywanie i manipulacja danymi klientów oraz przechowywanie i manipulacja danymi z jakiejś symulacji). Jednak sposób przechowania i modyfikacji tych danych może być identyczny.

Chcąc zatem zaprojektować rozwiązanie pozwalające na (bezpieczną, np. typowo[2]) pracę z kodem, gdzie wspomniane typy nie są w jakikolwiek sposób wymienne, należałoby zduplikować potrzebną architekturę, która miałaby na celu przechowywanie i modyfikację tych danych. Należałoby stworzyć dwie kolekcje oraz dwa algorytmy służące do odwracania tych kolekcji.

---

[1] W tym kontekście termin „obiekt" jest używany w ten sam sposób, jak w standardzie języka C++, czyli dotyczy on również zmiennych prymitywnych z języków takich jak Java czy C#.
[2] Tutaj – *związaną z typami danych*.

Już przy pierwszym takim rozdzieleniu danej potrzeby na obsługę dwóch przypadków biznesowych napotyka się na problem duplikacji kodu, który jest jednym z głównych przewinień procesu wytwarzania oprogramowania, o którym jest więcej mowa w dalszej części pracy.

2.3 Problem duplikacji kodu

Podstawowym problemem duplikacji kodu jest ogromna trudność w utrzymaniu i pielęgnowaniu (ang. *maintainability*) systemu. „Software", czyli angielski termin tłumaczony na „oprogramowanie" składa się w pierwszym swoim członie ze słowa „soft", czyli „miękki". Oprogramowanie jest elastyczne i się zmienia. Należy brać to pod uwagę projektując rozwiązania. Niektóre aplikacje działają nieustannie od dekad. W tym czasie musiały również się zmieniać, aby sprostać nowym oczekiwaniom i potrzebom.

Program, który składa się z wielu rozproszonych elementów koncepcyjnie odpowiedzialnych za to samo, jest trudny w utrzymaniu. Zmiana danej koncepcji wiąże się z, często taką samą, zmianą w wielu miejscach w kodzie. Niesie to za sobą wiele problemów, takich jak:

- trudność w określeniu, czy wszystkie miejsca odpowiadające za implementację danej koncepcji zostało jednolicie i konsekwentnie zmienione;
- niepotrzebnie marnowany czas na wprowadzenie wielu identycznych zmian, gdy powinno się je wprowadzić raz;
- trudność w identyfikowaniu, czy powtórzone fragmenty kodu są częścią tej samej logiki, czy jednak mogą być rozwijane w odrębny sposób.

Problem powtarzalności kodu jest poruszany w wielu uniwersalnych zasadach dotyczących praktyk programistycznych, na przykład w DRY – ang. *Don't Repeat Yourself* [4]. Mimo że autorzy tego terminu zaznaczają, że ma on znacznie szersze zastosowanie i niesie za sobą więcej niż tylko wiadomość „*nie kopiujcie i wklejajcie fragmentów kodu*" [5], to właśnie z tym procesem, a raczej z przestrzeganiem przed nim, DRY jest najczęściej kojarzone.

Jak zatem zaprogramować podobną funkcjonalność dla dwóch, niekompatybilnych, kompletnie niezwiązanych ze sobą typów danych? Z reguły wystarczy zdefiniować zestaw uniwersalnych operacji, które muszą być obecne, aby dana funkcjonalność mogła wykonać swoją pracę.

W przypadku opisanym wyżej jest to przechowywanie jednego z dwóch (lub dowolnego) typów danych w sposób pozwalający na ustanowienie pewnej kolejności. Następnie trzeba zdefiniować algorytm odwracający kolejność elementów.

Implementację kolekcji, która pozwala przechowywać elementy dowolnego typu nazwiemy *implementacją generyczną*. Implementację algorytmu, który pozwala odwrócić kolejność elementów nazwiemy tak samo. Tak powstałe definicje terminów *generycznej kolekcji, generycznego kontenera* czy *generycznego algorytmu* będą używane w dalszej części pracy.

Aby stworzyć generyczną implementację, należy zdefiniować zachowania, które musi wspierać typ danych, których ta implementacja dotyczy. Błędnym byłoby założyć, że generyczny kod musi działać z dowolnym typem danych. Jeżeli nie da się zamienić dwóch obiektów miejscami, to nie da się odwrócić

kolekcji tych obiektów. Nie czyni to jednak danej implementacji niegeneryczną. Tak samo generyczna kolekcja, która przechowuje wszystkie swoje elementy w kolejności zgodnej z jakimś porządkiem (np. zawsze posortowane) wciąż jest generyczna, nawet jeżeli nie będzie wspierała typu, dla którego żaden porządek nie istnieje.

Problem duplikacji kodu można zatem rozwiązać korzystając z narzędzi programowania generycznego. Nie są one jednak trywialne. Różne języki programowania pozwalają określić cechy, które dany typ spełnia, w różny sposób. Jako że różnice istnieją, to z pewnością mają one podstawy w fakcie, że nie istnieje jedno, idealne i uniwersalne rozwiązanie. Implikuje to istnienie potrzeby wyboru pomiędzy narzędziem, które lepiej nadaje się do jednej rzeczy, a gorzej do drugiej.

Należy zauważyć podobieństwo między motywacją ku istnieniu wielu mechanizmów programowanie generycznego, a motywacją ku istnieniu wielu różnych języków programowania. Język programowania to również nic innego jak tylko narzędzie. Dany język może być idealnym rozwiązaniem dla danego problemu, ale kompletnie nie nadawać się do użytku w przypadku innego rodzaju aplikacji.

2.4 Trudności w wytwarzaniu generycznego oprogramowania

Aby lepiej zilustrować różnice między podejściami do programowania generycznego, zostaną przedstawione przykłady generycznych implementacji z trzech, relatywnie odległych, a przynajmniej w kwestii narzędzi do programowania generycznego, języków programowania: Javy, C++ i Pythona.

Należy uzasadnić wybór akurat tych trzech języków. Powód jest dosyć prosty – każdy z nich zupełnie inaczej podchodzi do programowania generycznego.

Java i C++ należą tej samej rodziny języków – przynajmniej historycznie [3]. Wywodzą się z gałęzi ewolucji, która dotyczy języków takich jak CPL, BCPL, B, C, SIMULA 67 czy ALGOL 68. Cechują się podobną składnią dotyczącą deklaracji typów, zmiennych oraz samej struktury programu.

Języki te obrały jednak zupełnie różne strategie dotyczące wspierania programowania generycznego. Java wspiera ten paradygmat za pomocą *typów generycznych* (ang. *generics*) oraz *interfejsów* (ang. *interfaces*). C++ wprowadził szablony (ang. *templates*) oraz, wraz z C++20, Koncepty[3] (ang. *Concepts*). Ich różnice i wpływ na jakość programowania generycznego będą omówione w kolejnych podrozdziałach.

Python wyróżnia się wśród wybranej trójki. Współdzieli jeden korzeń drzewa genealogicznego z językiem C++, ale druga część jego historii opiera się na rozwoju takich języków jak MODULA-3 czy nowsze wersje FORTRANa (wersja 77) [3]. Python jest językiem dynamicznie typowanym. To znaczna różnica w kontraście do Javy czy C++, które są językami statycznie typowanymi.

[3] W kontekście niniejszej pracy termin „Koncept" (pisany dużą literą) referuje do narzędzia / bytu wprowadzonego w standardzie C++20, a termin „koncept" (pisany małą literą) jest synonimem terminu „idea".

2.4.1 Przykład programowania generycznego w języku Python

Oryginalnie przedstawiony problem, którego rozwiązaniem było użycie programowania generycznego, opierał się na niekompatybilności typów, co skutkowało – najczęściej – błędami kompilacji. Jak zatem patrzeć na Pythona, który nie kładzie takiego nacisku na konkretny typ obiektu (który dla programisty jest często w ogóle nieznany)?

Wbrew pozorom – znacznie prościej niż na Javę czy C++. Listing 4 przedstawia przykład funkcji zaimplementowanej w języku Python, która sortuje kolekcję.

Listing 4: Implementacja algorytmu sortowania bąbelkowego w Pythonie 3.10.

```
1 def bubble_sort(arr):
2     n = len(arr)
3
4     for i in range(n):
5         for j in range(0, n - i - 1):
6             if arr[j] > arr[j + 1]:
7                 arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Patrząc na te implementację, należy zwrócić uwagę, że nie występuje tam użycie jakiegokolwiek narzędzia specyficznego dla programowania generycznego. Języki dynamicznie typowane, można by rzec, z założenia tworzą konstrukcje generyczne.

W czasie wykonania (ang. *runtime*) interpreter Pythona po prostu sprawdza, czy da się wywołać funkcję `len()` na argumencie `arr`. Później sprawdza, czy obiekt `arr` wspiera indeksowanie (np. przy `arr[j]`). Następnie sprawdza, czy elementy tej kolekcji da się porównać za pomocą operatora `>`. Na koniec sprawdza, czy elementy `arr`, do których można się dostać poprzez indeksowanie, można nadpisać.

Nie jest istotne, czy `arr` to lista, czy dowolna inna kolekcja wspierająca indeksowanie. Nie jest istotne, czy elementy `arr` to obiekty reprezentujące klientów biznesowych, czy wartości jakiejś symulacji. Istotne jest jedynie to, aby można było wykonać wymienione wyżej operacje.

W tym miejscu warto jeszcze wspomnieć o tym, że w obecnych czasach wiele języków dynamicznie typowanych jest wzbogacane o rozszerzenia pozwalające na wprowadzenie dodatkowych informacji o typie danych podczas fazy kompilacji lub preprocesowania. Popularnym tego typu rozszerzeniem jest TypeScript, który jest swojego rodzaju nakładką na język JavaScript.

Python również doczekał się podobnych rozszerzeń – w postaci biblioteki `typing`. Pozwala ona na zmianę sygnatury funkcji z `def bubble_sort(arr)` na `def bubble_sort(arr: List[int]) -> List[int]`. Może się wydawać, że wtedy funkcja będzie działała tylko dla liczb całkowitych. Nie jest to jednak prawda.

Nie bez powodu to narzędzie (moduł `typing`) jest nazywane *wskazówkami typów* (ang. *type hints*), zamiast – po prostu – *typami*. Podanie złego typu poskutkuje jedynie ostrzeżeniem wygenerowanym przez

środowisko (jeżeli w ogóle wspiera ono generowanie takich ostrzeżeń), a nie błędem, jak by to było w przypadku użycia złego typu w języku TypeScript.

Omówienie dynamicznego typowania jako narzędzia używanego do programowania generycznego zostało przedstawione w dalszej części pracy.

2.4.2 Przykład programowania generycznego w języku Java

Język Java jest statycznie typowany. Oznacza to, że stworzenie obiektu musi zostać opatrzone deklaracją typu, który się nie zmienia w czasie działania programu. Typ wysyłany do metody musi się perfekcyjnie zgadzać z typem, który metoda przyjmuje (z wyjątkiem domyślnej konwersji przy polimorficznych hierarchiach, która aplikuje się też do fragmentu ze zmianą typu). Sprawia to, że implementacja algorytmu czy struktury danych będzie ograniczona do pracy na jednym typie danych (lub na danej hierarchii dziedziczenia).

W rozdziale **2.2** zostało użyte stwierdzenie „*Hierarchie polimorficzne są z reguły relatywnie płytkie.*” Uzasadnieniem tego stwierdzenia jest idea tworzenie łatwych do rozwijania i pielęgnowania hierarchii, które odpowiadają za daną, **jedną** funkcjonalność. Wiele renomowanych autorytetów zajmujących się architekturą oprogramowania zgadza się z tą tezą, patrz np. [6], [7], [8] i [9].

Warto przyjrzeć się tematowi polimorfizmu w Javie, ponieważ, poniekąd łamiąc wyżej przytoczoną wytyczną, język ten zmusza wszystkie typy klasowe do przynależności do jednej, ogromnej hierarchii dziedziczenia. Wszystkie klasy w Javie pośrednio lub bezpośrednio dziedziczą z klasy `Object`. Oznacza to, że przechowując obiekty (i wykonując na nich jakąś pracę) za pomocą referowania do nich przez typ `Object`, logika może efektywnie współdziałać z każdym typem.

Niestety nie jest to w pełni prawda. W przeciwieństwie do Pythona, Java nie pozwoli na wykonanie arbitralnych operacji na obiekcie, których poprawność nie będzie zweryfikowana w czasie wykonania. Przekłada się to na następujący wybór, gdy architektura przechowuje referencje typu `Object`:

1. Ograniczenie się wyłącznie do operacji możliwych do wykonania na zmiennych, o których nic nie wiadomo (poza tym, że dziedziczą po `Object`). Tych operacji jest wyjątkowo mało, choć pozwoliłoby to już na przechowywanie dowolnych typów danych w generycznej kolekcji oraz na zaimplementowanie generycznego algorytmu jej odwracania.

2. Korzystanie z rzutowania (ang. *explicit casts*), aby obejść statyczne typowanie języka. W ten sposób programista zapewnia, że „pod spodem” znajduje się obiekt bardziej konkretnego typu pozwalający na więcej operacji. Niestety, takie podejście jest narażone na ogrom błędów – od wyjątków, przez brak dobrego wsparcia innych narzędzi wspomagających pracę programisty, kończąc na trudnych do czytania i analizy fragmentów kodu.

Aby obejść te problemy, Java w wersji 5 (wydanej w 2004 roku) wprowadziła typy generyczne. Efektywnie są one ustandaryzowanym mechanizmem automatyzującym rzutowanie, co rozwiązuje problemy omówione powyżej.

Typy generyczne zostały dodatkowo wzbogacone o możliwości ich ograniczania (ang. *constraining*) za pomocą słów kluczowych `extends` lub `super`. Wbrew intuicyjnym skojarzeniom z tym terminem,

ograniczanie typów generycznych prowadzi do możliwości wykorzystania ich w **szerszym** kontekście. Gdy typ jest nieograniczony, to program nie wie o typie nic poza tym, że dziedziczy po klasie `Object`. To bardzo niewiele. Jeżeli jednak typ zostanie ograniczony jako dziedziczący po innej klasie lub implementujący jakiś `interface`, to Java będzie „wiedziała”, że na tych obiektach można wywołać operacje unikalne dla danej hierarchii klas czy interfejsu.

Pierwsze przedstawienie typów generycznych z Javy można zobaczyć w Listing 3, a przykład użycia przy wprowadzanym już algorytmie sortowania bąbelkowego w Listing 5.

Listing 5: Implementacja algorytmu sortowania bąbelkowego tablicy w Javie w wersji 5+.

```
1  public static <T extends Comparable<? super T>>
2  void bubbleSort(T[] arr) {
3      int n = arr.length;
4
5      for (int i = 0; i < n - 1; i++) {
6          for (int j = 0; j < n - i - 1; j++) {
7              if (arr[j].compareTo(arr[j + 1]) > 0) {
8                  T temp = arr[j];
9                  arr[j] = arr[j + 1];
10                 arr[j + 1] = temp;
11             }
12         }
13     }
14 }
```

Najistotniejszym fragmentem tego kodu jest wprowadzenie (ograniczonego) typu generycznego `T` dzięki następującej składni: `<T extends Comparable<? super T>>`. Na jego podstawie przyjmowany jest argument `arr` – tablica elementów typu `T`. Typ `T` jednak nie może być zamiennikiem (ang. *placeholder*) kompletnie dowolnego, istniejącego typu. Elementy tablicy muszą implementować interfejs `Comparable` w taki sposób, aby można je było porównywać między sobą.

Interfejsy (jako część metadanych programu Javowego, tworzone dzięki słowie kluczowemu `interface`) są jednym z najważniejszych elementów tego języka. Ujednolicają narzędzia używane do projektowania architektury programów w Javie. Dokładniejsze omówienie interfejsów oraz ich porównanie z alternatywnymi narzędziami, zwłaszcza w kontekście programowania generycznego, zostało omówione w dalszej części pracy.

Warto jeszcze zaznaczyć, że, jak już zostało to wspomniane wcześniej, typy generyczne w Javie są efektywnie jedynie automatyzacją mechanizmu rzutowania. Rzutowanie to, w swojej podstawie, operuje na typie `Object`. W związku z tym, że w tym języku istnieje silne rozróżnienie między typami prymitywnymi, a złożonymi, Java nie wspiera programowania generycznego dla typów takich jak `int` czy `double`.

Dlatego Java w wersji 5, wraz z typami generycznymi, wprowadziła mechanizmy pozwalające na łatwiejszą, często automatyczną, konwersję między prymitywami a typami opakowującymi (ang. *wrappers*). Mechanizmy te zostały oryginalnie nazwane:

- *boxing* – opakowywanie / konwersja typu prymitywnego na typ klasowy, np. `int` na `Integer`;

- *unboxing* – rozpakowywanie / konwersja typu klasowego na typ prymitywny, np. `Double` na `double`.

O ile ich użyteczność jest wysoka, to nawet gdy wzbogacimy typy generyczne o wyżej wspomniane mechanizmy, wciąż niemożliwym będzie tworzenie w pełni generycznego kodu w tym języku. Typy generyczne zwyczajnie nie działają z prymitywami. W związku z tym nie jest możliwym stworzenie obiektu typu `ArrayList<int>` czy `HashMap<char, boolean>`.

Należy jeszcze wspomnieć, że proces polegający na automatyzacji rzutowania przy typach generycznych efektywnie wymazuje informację o faktycznym typie danego obiektu. Oczywiście mowa o informacji dostępnej w czasie kompilacji. W czasie wykonania wciąż można użyć mechanizmu refleksji czy dynamicznej informacji typu, aby otrzymać informacje o dokładnym typie obiektu. Proces wymazywania typu po angielsku nazywa się *type erasure*.

2.4.3 Przykład programowania generycznego w języku C++

Język C++, podobnie jak Java, jest statycznie typowany. Tworzenie obiektów jest opatrzone deklaracją typu, który nie może się zmienić podczas działania aplikacji. Język ten również wspiera dziedziczenie i polimorfizm, ale narzędzia te nie są tak silnie zintegrowane z całym ekosystemem jak w przypadku Javy. W C++ nie mamy do czynienia z „drzewem klas", czyli z hierarchią, która wywodzi się od jednej, wspólnej nadklasy (w przypadku Javy – klasy `Object`). Mamy jednak do czynienia z „lasem klas", czyli z hierarchią, gdzie klasy mogą po sobie dziedziczyć (nawet wielokrotnie, dzięki wielodziedziczeniu, którego Java nie wspiera), ale nie muszą. Nie ma jednego, wspólnego `Object`u.

Dodatkowo należy również zwrócić uwagę na rozróżnienie semantyk obydwu języków dotyczących odnoszenia się do obiektów. W języku Java (oraz Python) mamy do czynienia z semantyką referencji[4], a w języku C++ semantyką wartości.

Semantyka wartości języka C++ sprawia, że chęć odnoszenia się do oryginalnego obiektu z poziomu innego fragmentu kodu (np. z funkcji) musi być jawnie zaznaczona. W takich przypadkach należy skorzystać z referencji lub wskaźników.

Referencje i wskaźniki są, formalnie, osobnymi bytami w hierarchii typów. Jedną z konsekwencji płynącą z tego jest brak możliwości oparcia podstawowych operacji generycznych na wspólnej nadklasie (jak to miało miejsce w Javie), zwłaszcza biorąc pod uwagę brak istnienia takowej w języku C++.

Dedykowanym narzędziem do programowania generycznego w C++ są szablony (ang. *templates*). Służą one, podobnie jak typy generyczne w Javie, do wprowadzania nazw, które mogą reprezentować wiele różnych typów. W przeciwieństwie jednak do typów generycznych, szablony w pełni wspierają generyczne programowanie. Nie istnieją ograniczenia w postaci braku możliwości wspierania typów prymitywnych.

[4] Istnieje pewne odchylenie od normy – język Java czy Python respektują semantykę referencji przy obiektach złożonych, ale w przypadku prymitywów operacje cechują się semantyką wartości. Jest to, zdaniem autora pracy, niekonsekwentność, która, o ile ma poparcie w architekturze języka, może wprowadzać wiele niejednoznaczności i trudności w rozumowaniu kodu, a zwłaszcza generycznego.

Listing 6[5] przedstawia przykład sortowania bąbelkowego w języku C++.

Listing 6: Implementacja algorytmu sortowania bąbelkowego wektora w C++ w wersji 11.

```
 1  template <typename T>
 2  auto bubbleSort(std::vector<T>& vec) -> void {
 3      auto n = vec.size();
 4
 5      for (auto i = 0; i < n - 1; i++) {
 6          for (auto j = 0; j < n - i - 1; j++) {
 7              if (vec[j] > vec[j + 1]) {
 8                  std::swap(vec[j], vec[j + 1]);
 9              }
10          }
11      }
12  }
```

Linijka `template <typename T>` mówi o tym, że następna deklaracja będzie szablonem, w którym widoczny będzie parametr szablonowy `T`. Mówiąc najprościej, nazwa `T` zastępuje dowolny typ, którego użytkownik generycznego kodu akurat potrzebuje użyć.

Implementacja w języku C++ w pewien sposób łączy niektóre z cech zaprezentowanych w przykładach z Javy i Pythona. Z jednej strony widać, że C++ wymaga deklaracji typu generycznego, aby dopasować użycie algorytmu do statycznego systemu typowania, czyli dopasować `T` tak, aby pasował do przekazywanych obiektów. Dokładnie taka sama sytuacja miała miejsce w Javie.

Z drugiej strony widać, że typ `T` nie jest jakkolwiek ograniczony – nie musi być. Język C++ dopasuje go do aktualnie potrzebnego wykorzystując zasady dedukcji typu. Nie trzeba z góry określać wymagań dotyczących zakresu akceptowalnych typów. Jeżeli dopasowany typ nie będzie wspierał określonych operacji (np. nie będzie można porównać obiektów tego typu za pomocą operatora `>` lub zamienić ich miejscami), program nie zadziała – tak, jak w przypadku przykładu kodu w Pythonie[6].

Pod pewnymi względami można powiedzieć, że wersja z szablonami z C++ jest połączeniem najlepszych cech obydwu światów (Javowego i Pythonowego). Wciąż sprawdza poprawność operacji w czasie kompilacji (jak w przypadku Javy), ale nie wymaga ani ograniczania typów szablonowych, ani nie wprowadza wymogu deklaracji implementowania danego interfejsu, czego też nie robi Python.

W dalszej części pracy te plusy i minusy zostaną dokładniej omówione i ponownie przeanalizowane.

---

[5] Warto zwrócić uwagę, że jest to nie do końca poprawna implementacja algorytmu sortowania bąbelkowego z uwagi na błędny typ wybrany do indeksowania kolekcji. O ile w Javie i w Pythonie używa się do tej operacji typu `int`, to w C++ wielkość kontenerów zwykle reprezentuje się za pomocą `std::size_t`, który najczęściej jest aliasem do typu `unsigned long long int`. W C++23 został wprowadzony literał `uz`, który by ułatwił poprawną implementację tego algorytmu, ale z uwagi na marginalne znaczenie w kontekście tego, co przykład ma ilustrować oraz mając na uwadze konsekwentność przykładów, użycie go zostało pominięte.

[6] Z tą różnicą, że kod w Pythonie się uruchomi i poskutkuje podniesieniem wyjątku w momencie próby wykorzystania niedostępnej operacji, a kod w języku C++ się w ogóle nie skompiluje.

3 Historia języka C++ w kontekście ogólnie pojętego programowania generycznego

Praca ta koncentruje się w szczególności na nowych elementach języka C++, dzięki którym programowanie generyczne może być łatwiej, bezpieczniej i szerzej aplikowane. W celu dokładnego zrozumienia potrzeby, która zrodziła ww. narzędzia, dobrze jest przyjrzeć się dokładniej zarówno historii rozwoju jak i użyteczności języka C++. Dodatkowo, czytając poniższe rozdziały, należy pamiętać, że głównym narzędziem programowania generycznego są szablony.

Kod zaprezentowany w Listing 6 dotyczy standardu C++11, głównie z uwagi na użycie słowa kluczowego `auto`, które zostało szerzej opisane w dalszej części pracy. Identyfikacja konkretnego standardu ma jednak jeszcze jedno, dodatkowe znaczenie – wskazuje na fakt, że różne narzędzia w języku pojawiają się w różnym czasie. Tak jak typy generyczne w Javie, szablony – tak kluczowy element programowania w C++ współcześnie – wcale nie były częścią języka od samego początku.

Język C++ narodził się pierwotnie jako „C z klasami" (ang. *C with classes*) przez Bjarne Stroustrupa w 1979 roku. Po 5 miesiącach pracy doczekał się swojej pierwszej implementacji [10]. Od tamtego momentu nieustannie ewoluuje, choć te szczegóły tego procesu zdecydowanie różnią się od tych, które pierwotnie wprowadziły pierwsze zmiany w języku.

Jest to narzędzie wykorzystywane wszędzie tam, gdzie istotnym jest połączenie wysokiego poziomu abstrakcji z maksymalną wydajnością. C++, dzięki tym cechom, szybko zdobył wielu zwolenników, co przyczyniło się do jego nie tylko biznesowego, ale również i dydaktycznego sukcesu. W wielu szkołach czy uczelniach jest on nieodzownym elementem.

Ewolucja tego języka jest tematem, który doczekał się wielu publikacji, a nawet dedykowanej książki [10]. Jednak szczegóły samej tej ewolucji nie są istotne dla niniejszej pracy. W przypadku gdy wspomniany proces będzie znaczący do poparcia przedstawionych tez lub argumentów, odpowiednie wątki zostaną przytoczone i omówione.

Wśród historii standardów języka C++ (o których mowa w detalach w następnym podrozdziale) można wyróżnić cztery epoki[7]:

- Czas pre-standaryzacji. Pierwotny pomysł, projekt i praca Bjarne Stroustrupa nad nowym narzędziem w Bell Laboratories. 1997-1998.
- Pierwszy standard i jego poprawki. 1998-2011.
- Nowoczesny (ang. *modern*) C++. 2011-2022.
- Kompletny C++. (2022-obecnie).

Prawie każda epoka niesie za sobą znaczne zmiany i usprawnienia w kontekście programowania generycznego, dlatego zostaną one omówione w kolejnych podrozdziałach. Warto jeszcze jednak zwrócić

[7] Nie należy mylić tego terminu z *epokami* rozumianymi jako formalnymi, modularnymi konfiguracjami dotyczącymi znaczenia i zachowania kodu w częściach systemu zależnymi od wybranych opcji [10].

uwagę na termin *kompletny C++*. Brzmi on jak kolokwialne określenie sytuacji, w której żaden język programowania nie powinien nigdy się znaleźć. Język programowania, który nie ewoluuje, podlega procesowi stagnacji i zostaje wypierany przez inne narzędzia. Ginie. Lub gorzej – jest sztucznie utrzymywany przy życiu z uwagi na ogromne koszty, które dana organizacja musiałaby ponieść, gdyby pogodziła się z faktem, że dane narzędzie trzeba przepisać na inny język.

Dlatego *kompletność* nie oznacza tutaj *perfekcyjności*, czyli stanu, którego „nie należy ruszać". Reprezentuje ona stan, w którym język C++ znajduje się na zasadzie wspierania wszystkich operacji i posiadania wszystkich narzędzi, które jego twórca chciał w nim oryginalnie zawrzeć. Mowa tutaj między innymi o Konceptach, Modułach (ang. *Modules*) czy Korutynach (ang. *Coroutines*).

3.1 C++ przed pierwszym oficjalnym standardem

Szablony nie istniały w języku od samego początku, jak już zostało to wcześniej wspomniane. Pierwotnie, zostały one pominięte z uwagi na brak wystarczającej ilości czasu na precyzyjne, dokładne i pełne badania pozwalające na stworzenie odpowiedniej jakości specyfikacji tego narzędzia [10].

W języku C, na którym buduje się język C++, pewną formą programowania generycznego są makra (ang. *macros*). Makro to bardzo prosta, lecz jednocześnie bardzo potężna dyrektywa preprocesora, która polega na prostej zamianie tekstu. Najpopularniejszymi (ale też i sprawiającymi najwięcej problemów) są makra `MIN` oraz `MAX`, zdefiniowane w sposób przedstawiony w Listing 7.

Listing 7: Definicja makr `MAX` oraz `MIN` w języku C i C++.

```
1  #define MIN(a,b) ((a) < (b) ? (a) : (b))
2  #define MAX(a,b) ((a) > (b) ? (a) : (b))
```

Ich funkcjonalność jest wręcz trywialna. Wszędzie tam, gdzie zostaną użyte tokeny reprezentujące ciąg znaków MIN lub MAX, po których zostaną umieszczone nawiasy obejmujące dwie grupy znaków oddzielone przecinkiem, fragmenty te zostaną zastąpione definicjami makra.

Listing 8 prezentuje przykład działania ww. makr.

Listing 8: Przykład użycia makr `MAX` oraz `MIN`.

```
 1  #define MAX(a, b) ((a) > (b) ? (a) : (b))
 2  #define MIN(a, b) ((a) < (b) ? (a) : (b))
 3
 4  auto main() -> int {
 5      auto value1 = 10;
 6      auto value2 = 20;
 7
 8      auto const max = MAX(value1, value2);
 9      auto const min = MIN(value1, value2);
10  }
```

Na pierwszy rzut oka można nie zauważyć żadnych problemów. Wartością stałej `max` jest `20`, a `min` `10`. Tak, jak można się było tego spodziewać. Co więcej – niezależnie od tego, czy typem `value1` i `value2` jest `int`, `double` czy `float` (i czy w ogóle te typy są takie same), makra dalej zadziałają poprawnie. Porównają te wartości i zwrócą kolejno większą lub mniejszą wartość.

Niestety, istnieje ogromny minus w sposobie działania tych narzędzi. Nie robią one nic innego jak tylko zastępują tekst. Oznacza to, że użycie dowolnego wyrażenia z widzialnymi efektami ubocznymi (ang. *side-effects*) będzie wywołane nie raz – jak się może to wydawać – tylko tyle razy, ile zostanie ono użyte w definicji („rozwinięciu”) makra. Takim wyrażeniem jest na przykład preinkrementacja.

Listing 9: Przykład otrzymania błędnej wartości z powodu użyciu makra i preinkrementacji.

```
1  auto value      = 10;
2  auto const max = MAX(++value, 0);
```

Oczekuje się, że zmienna `max` z Listing 9 będzie miała wartość `11`. Makro `MAX` w końcu otrzymuje dwa, proste argumenty – wynik operacji preinkrementacji zmiennej `value` (czyli zmienna `value` zmienia swoją wartość na `11`, a potem jest wysyłana do makra) oraz literał `0`.

Niestety, w powyższym rozumowaniu pojawił się fundamentalny błąd. Do makra `MAX` nie jest wysyłany wynik preinkrementacji. Do makra `MAX` jest wysyłany tekst, którego interpretacja leksykalna oznacza preinkrementację zmiennej `value`. Oznacza to, że preprocesor podmieni tekst z drugiej linijki na fragment kodu zaprezentowany w Listing 10.

Listing 10: Rozwinięte makro `MAX` demonstrujące problem używania wyrażeń z efektami ubocznymi.

```
1  auto const max = ((++value) > (0) ? (++value) : (0));
```

Jak widać, operacja preinkrementacji nie tylko pojawiła się dwa razy w kodzie. Ona została **wykonana dwa razy**. Stąd wartość zmiennej `max` wynosi nie `11`, a `12`.

Jasno zatem widać, że używanie makr do programowania generycznego nie jest najlepszym pomysłem. O ile pozwalają one kompletnie zignorować typ zmiennych i operować na surowym tekście kodu, to wcale nie jest to coś, do czego docelowe narzędzie powinno dążyć. Programowanie generyczne wciąż powinno wspierać bezpieczeństwo typowania i integrować się w język, a nie operować na generowaniu tekstu, który będzie interpretowany jako kod.

Jak można przeczytać w [10], Bjarne Stroustrup zaznacza, że makra nie koegzystowały spójnie z innymi, kluczowymi elementami C++. Pierwotnie były one używane do parametryzowania kolekcji, ale w tych kontekstach wychodziły na świat inne problemy tego rozwiązania – makra nie respektują ograniczeń widoczności bloków (ang. *scope*) oraz nie były wystarczająco wspierane przez narzędzia wspomagające prace programisty. Przykładowo, kompilator nie wie o istnieniu makr. Dla niego kod źródłowy C++ wygląda identycznie, niezależnie od tego, czy powstał w wyniku rozwinięcia kilku makr, czy nie.

3.1.1 Wprowadzenie szablonów

Problemy omówione w poprzednim rozdziale zrodziły potrzebę stworzenia nowego, lepiej przystosowanego narzędzia do programowania generycznego. Tak zrodziły się szablony języka C++. Były one manifestacją odpowiedzi „tak" na poniższe pytania:

- „Czy parametryzacja może być prosta w użyciu?
- Czy obiekty o typach sparametryzowanych typach mogą być użycie tak samo efektywnie jak obiekty typów zdefiniowanych „ręcznie"?
- Czy istnieje możliwość zintegrowania generalizacji typów parametryzowanych do C++?
- Czy typy sparametryzowana mogą być zaimplementowane w taki sposób, że czas kompilacji i łączenia[8] będzie porównywalny do tego, który można osiągnąć w systemie, który nie wspiera takich typów?
- Czy taki system może być prosty i przenośny[9]?"

~ Bjarne Stroustrup [10], rozdział 15.2.

Kluczowe idee dotyczyły:

- przyjaznej notacji;
- wydajności czasu wykonania;
- bezpieczeństwa typów.

Z uwagi na to, mając do wyboru wzorowanie się na mechanizmie Smalltalka, które opierało się na dziedziczeniu i dynamicznym typowaniu lub na mechanizmie Clu, które opierało się na statycznym typowaniu i obsługiwaniu generycznego typu reprezentującego typy danych, decyzja była relatywnie oczywista – szablony zostały stworzone jako zbiór mechanizmów, które generują kod pracując na parametrach szablonowych. Można je nazwać typami typów, zamiennikami typów czy nawet generatorami kodu, dzięki czemu nie rezygnuje się z przewag statycznego typowania – tak jak w przypadku Clu.

Oryginalnie szablony były relatywnie wybrakowane. Brakowało w nich wielu elementów, takich jak niesprecyzowana liczba argumentów czy specjalizacje, które zostały dodane w późniejszych etapach ewolucji języka. Niemniej jednak, C++ otrzymał narzędzie, które do dziś pozwala pisać bardzo potężny, elastyczny, ale również i bezpieczny kod[10].

---

[8] ang. *linking*.
[9] ang. *portable*.
[10] Z uwagi na to, że szablony efektywnie generują kod, a nie go „ugenerityczniają" (jak typy generyczne w Javie), można się spierać, czy poprawnym terminem jest tutaj „pisanie kodu". Mimo wszystko celem programowanie jest stworzenie narzędzia. Mechanizm działania szablonów współgra na tyle ściśle z resztą elementów języka C++, że, w przeciwieństwie do makr, warto rozważyć zaakceptowanie skrótu traktującego o tym, że tworzenie szablonów to „pisanie kodu", a używanie makr to „generowanie kodu", mimo że efektywnie obydwa te elementy w pewien sposób kod generują.

Warto wspomnieć o swojego rodzaju legendzie, która stanowi, że pierwotne dodanie szablonów do języka C++ było obwite w coraz to większe zaskoczenia osób pracujących przy nim. Podobno wiele elementów, na które pozwoliły szablony (efektywnie będące Turing-kompletnym metajęzykiem) były niezaplanowane i „wyszły przypadkiem". O ile faktycznie autor pracy nie znalazł żadnych źródeł traktujących o tym, że ta kompletność była jednym z celów do osiągnięcia przez narzędzie szablonów, to tak samo nie znalazł nic o tym, że wiele ich użytecznych objawiło się „przypadkiem".

3.2 Pierwszy standard i jego poprawki

Język C++ jest ustandaryzowany. Pierwszy jego standard ukazał się w roku 1998. Zawierał on wszystkie elementy języka, opisane formalnym tonem w dokumencie standaryzacyjnym. Dowolne zmiany muszą być realizowane przez formalne wnioski (ang. *proposals*) lub raporty wadliwości (ang. *defect reports*) składane komitetowi standaryzacyjnemu.

Przez pierwsze 13 lat (1998 – 2011) swojej formalnej egzystencji, język C++ nie przechodził większych zmian. Drugim jego standardem była wersja 03, której nazwa referuje do roku 2003. Jest ona okrzyknięta „małą korektą", bo składała się głównie z poprawek w postaci raportów wadliwości. Niewiele się zmieniło w tym okresie. Język przechodził przez okres stagnacji, który – jak się później okazało – był ciszą przez burzą pod postacią publikacji standardu C++11.

3.3 Nowoczesny C++

„*Modern C++*", czyli „Nowoczesny C++" to nazwa, którą określa się stan języka, sposób pisania w nim oraz narzędzia, które się pojawiły w C++ wraz ze standardem z 2011 roku. Zmiany były ogromne. Rdzenny język został wzbogacony o, między innymi:

- Słowo kluczowe `auto` pozwalające na deklarację zmiennej o wydedukowanym typie na podstawie inicjalizacji.
- Słowo kluczowe `constexpr`, które pozwoliło na bardziej naturalne wyrażanie obliczeń, które mogą (lub muszą) być wykonane w czasie kompilacji. Efektywnie pozwoliło na definiowanie kodu, który musiał być uruchamiany w czasie kompilacji, za pomocą „klasycznego" kodu C++. Zastąpiło część przypadków użycia metaprogramowania.
- `static_assert`. Pozwala na zdefiniowanie (ewaluowanego w czasie kompilacji) wyrażenia, które – jeżeli będzie miało wartość logiczną `false` – poskutkuje przerwaniem kompilacji.
- Lambdy, czyli prosty sposób na stworzenie obiektu o anonimowym typie z odpowiednim wsparciem dla używania go jako funkcja (obiekt funkcyjny). Zrewolucjonizowały użyteczność standardowych algorytmów dzięki zwiększeniu ich elastyczności. Potrzeba stworzenia „lokalnej funkcji" zmieniającą punkt dostosowania istniała jeszcze przed ideą szablonów w C++. Nawet funkcja `qsort()` z C przyjmuje swojego rodzaju generyczny komparator (wykorzystujący wskaźniki na `void` i wymagający rzutowania, co – jak już wiadomo – nie jest optymalną strategią osiągnięcia generyczności).

- Semantyki przenoszenia (ang. *move semantics*). Wprowadziły one nową kategorię wartości, która pozwoliła na precyzyjniejsze określanie stanu żywotności obiektu (czy obiekt jest tymczasowy, a jeśli nie, to czy nie powinno się go traktować, jakby taki był), dzięki czemu udało się osiągnąć bezpieczny typowo sposób na proste zwiększenie wydajności wielu konstrukcji.

- Słowo kluczowe `decltype()`, które pozwoliło na precyzyjne określenie typu danego wyrażenia. Szczególnie przydatne w generycznym kontekście.

- Szablony o zmiennej liczbie parametrów (ang. *variadic templates*). Pozwoliły na zdefiniowanie szablonów, które akceptują niesprecyzowaną liczbę parametrów. Pozwoliły na stworzenie takich typów jak krotka (ang. *tuple*). W pewien sposób dopełniły kompletność metaprogramowania i generalnie programowania generycznego w C++.

Biblioteka standardowa została rozszerzona między innymi o:

- Inteligentne wskaźniki (ang. *smart pointers*). Problem zarządzania zasobami (głównie pamięcią) był (i jest) w C++ dobrze znany. Mimo tego, że język szczyci się tak potężną cechą jak deterministyczna destrukcja (dokładnie sprecyzowane czas i miejsce destrukcji obiektu z możliwością sterowania tym procesem), która jest realizowana przez destruktory, pewne problemy są na tyle wszechobecne, że wprowadzenie rodziny wskaźników, które by automatyzują zarządzanie pamięcią, okazało się być kluczowe.

- Bibliotekę `<type_traits>`, która wspiera metaprogramowanie pozwalając na analizę cech typów generycznych.

Jest to tylko fragment elementów dodanych wraz z C++11. Każdy z nich jest jednak absolutnie niepomijalny i używany w większości aplikacji wykorzystujących ten język w tym standardzie.

Warto zwrócić uwagę na to, ile z tych elementów jest bezpośrednio lub pośrednio powiązane z programowaniem generycznym:

- `auto` pozwala na (między innymi) skrócony zapis deklaracji zmiennej typu, który może być bardzo skomplikowany z uwagi na charakterystykę programowania generycznego.

- `constexpr` i `<type_traits>` pozwalają na bardzo wygodne weryfikowanie cech typu, na którym ma pracować dany algorytm w kontekście generycznym, dzięki czemu można w bardziej naturalny sposób wybrać optymalną implementację, zamiast delegować te logikę do narzędzi metaprogramistycznych.

- `decltype()` jest swojego rodzaju dopełnieniem `auto` – pozwala na sprecyzowanie typu arbitralnego wyrażenia bez potrzeby materializowania obiektu. Przykładowo, można zbadać, jaki byłby typ wyniku wywołania danej funkcji i, za pomocą aliasu szablonu (również dodanego w C++11), referować do niego.

- Narzędzie *variadic templates* nie potrzebuje w tym miejscu żadnego komentarza. Brak możliwości tworzenia narzędzi pracujących na zmiennej liczbie parametrów jest ograniczeniem, którego nie powinno być. Alternatywne rozwiązanie korzystające z funkcji wariadycznych (ang. *variadic functions*) jest wysoce podatne na błędy.

Jak widać, język zmienił się dosyć drastycznie. Niekiedy można spotkać się z komentarzami, że „Programowanie w C++11 pozostawia wrażenie, jakby się programowało w zupełnie innym języku."

Niestety, pewne bardzo istotne elementy wciąż nie były obecne w języku. Programowanie generyczne wciąż wiązało się z nadużywaniem bardzo trudnych w obsłudze narzędzi, gdy chciało się osiągnąć maksimum wydajności i elastyczności. Szczegóły te będą omówione dokładniej w dalszej części pracy.

3.3.1 Zmiany praktyk programistycznych w ramach nowoczesnego C++

Nowoczesny C++ wprowadził nie tylko wiele ułatwień, wśród których programiści mogli przebierać do woli w poszukiwaniu elementów ułatwiających ich pracę, ale również i wiele nowych idiomów oraz (czasem kontrowersyjnych) reguł. W tym rozdziale za przykład posłuży idiom AAA, czyli *Almost Always Auto*, tłumaczony na język polski jako „prawie zawsze `auto`".

> "The C++ world is moving to *left to right* consistently."
>
> ~ Herb Sutter

Fenomenalne materiały Herba Suttera [11], [12] oraz Jonathana Boccary [13] stały się podwaliną adaptacji tego słowa kluczowego w kontekście nie tylko dedukcji typu ale i całego stylu programowania. Można też zauważyć, że wszystkie listingi kodu w języku C++ w niniejszej pracy respektują ten styl pisania. Język C++ nie ma oficjalnego standardu dotyczącego konwencji nazewnictwa (jak np. Java, gdzie, przykładowo, nazwy klas respektują **PascalCase**, a nazwy metod czy zmiennych **camelCase**).

Warto zatem się przyjrzeć i omówić AAA.

Zmienne lokalne w języku C++ można deklarować głównie na sposoby[11]:

- za pomocą składni `T t`, gdzie `T` to nazwa typu a `t` to nazwa zmiennej;
- za pomocą składni `auto t = I`, gdzie `t` to nazwa zmiennej, a `I` to wyrażenie inicjalizujące te zmienną.

Słowo kluczowe `auto` to – formalnie – specyfikator typu (ang. *type specifier*). Jego intuicyjne użycie (oraz jeden z głównych argumentów ku wprowadzeniu go do języka [14]) polega na dedukcji typu bazując na wyrażeniu inicjalizującym zmienną.

Jest to jednocześnie jeden z ważniejszych argumentów ku adaptacji stylu AAA. Aby `auto` „wiedziało", jakiego typu tworzona jest zmienna, deklarację należy też opatrzeć inicjalizacją. Sprawia to, że niwelowany jest problem niezainicjalizowanej zmiennej, zaprezentowany w Listing 11.

[11] Podział ten nie jest formalny. Pełna specyfikacja akceptowalnej gramatyki języka dla deklaracji jest bardzo obszerna, a jej podsumowanie można znaleźć na CppReference: https://en.cppreference.com/w/cpp/language/declarations.

Listing 11: Przykład błędu kompilacji spowodowanego pominięciem inicjalizacji zmiennej podczas korzystania z `auto`.

```
1  auto main() -> int {
2      int value1;
3      auto value2; // compilation error:
4                   // declaration of 'auto value2' has no initializer
5  }
```

W tym przykładzie zmienna `value1` jest niezainicjalizowana[12]. Oznacza to, że każda próba odczytania jej wartości (włącznie z próbami pośrednimi, np. poprzez skorzystanie z operatora `+=`) będzie skutkowała Niezidentyfikowanym Zachowaniem (ang. *Undefined Behavior*, *UB*) [15], chyba że w międzyczasie zostanie jej przypisana wartość (za pomocą operatora `=`).

Jest to łatwy do przeoczenia błąd, który może manifestować problemy w trudnych do identyfikacji miejscach i momentach. Stosowanie `auto` sprawia, że kompilator wymusza na programiście podanie wyrażenia inicjalizującego zmienną.

Kontrargumentem do tego przykładu są potrzeby stworzenia niezainicjalizowanego obiektu, którego celem będzie bycie buforem pamięci dla innych obiektów. W takim wypadku programista nie chce płacić kosztu inicjalizacji pamięci, która ma istnieć tylko po to, aby móc w niej tworzyć nowe obiekty. Przykład został pokazany w Listing 12.

Listing 12: Przykład użycia odpowiednio wyrównanego `std::array` jako bufora do danych.

```
 1  auto main() -> int {
 2      constexpr auto size = 10;
 3      using BufferType = std::array<std::byte, sizeof(int) * size>;
 4      alignas(int) auto intBuffer = BufferType();
 5
 6      auto data = reinterpret_cast<int*>(intBuffer.data());
 7      for (auto i = 0; i < size; i++) {
 8          *data++ = i;
 9      }
10  }
```

Listing 12 przedstawia przykład, gdzie zmienna `intBuffer` jest tworzona jako bufor pamięci dla 10 liczb całkowitych (`int`ów). Na skutek stosowania AAA jest tutaj, niestety, tracona wydajność. Tworzona tablica niepotrzebnie inicjalizuje okupowane przez siebie bajty pamięci zerami. To jest bezpośredni skutek stosowania AAA, który – najczęściej – jest dobrym zachowaniem. Tutaj jednak programista samemu chce nadpisać dane i nie potrzebuje pierwotnej inicjalizacji.

Autor pracy jest jednak zdania, że wcale nie jest to argument przeciwko AAA. Wręcz przeciwnie. To argument ku adaptacji tego stylu – z bardzo prostej przyczyny. Chęć pominięcia inicjalizacji jest jedynym

[12] Formalnie tak naprawdę ma miejsce tutaj proces inicjalizacji – Inicjalizacji Domyślnej (ang. *Default Initialization*), który w przypadku zmiennej typu `int` nic nie robi. Sytuacja byłaby jednak inna, gdyby zmienna była typu tablicowego lub klasowego.

argumentem przeciwko AAA w kontekście wydajności. Oznacza to, że zakładając adaptację tego stylu, gdy programista zobaczy, że zmienna jest deklarowana „starym stylem", to od razu wie, dlaczego. Autor kodu, który dany programista czyta, celowo chciał pominąć proces inicjalizacji. Stosowania AAA sprawia, że inicjalizacja jest domyślna – tak, jak być powinno. A w skrajnych przypadkach, gdy chce są ją pominąć, sama składnia deklaracji jest na tyle inna, że od razu widać, które obiekty są poddawane procesu pominięcia inicjalizacji.

Listing 13: Prezentacja tworzenia obiektu, którego inicjalizacja została celowo pominięta na tle domyślnie inicjalizowanych obiektów.

```
1  auto main() -> int {
2      auto someNumber = 10;
3      auto text       = std::string("abc");
4      alignas(int) std::array<std::byte, sizeof(int) * 10> intBuffer;
5      auto isWeatherGood = true;
6  }
```

Listing 13 jasno pokazuje, że linijka numer 4 wyróżnia się pośród innych. Nawet pomijając specyfikator `alignas()`, brak użycia auto przykuwa uwagę. Gdy AAA jest domyślnie stosowane, brak dostosowania się do niego wskazuje na jawne odejście od normy – najpewniej poparte dobrym powodem, który nie powinien zostać przeoczony.

Kolejnym argumentem za AAA jest konsekwentność. Czasem jedynym odpowiednim rozwiązaniem jest skorzystanie z `auto` przy deklaracji zmiennej, np. przy tworzeniu zmiennej inicjalizowanej lambdą. Lambdy, wprowadzone w C++11, mają unikalny typ, który, aby móc z niego skorzystać, musi być wydedukowany. Alternatywa w postaci wymazywania typu (podobnie jak w przypadku typów generycznych w Javie), na przykład korzystając z szablonu `std::function` może skutkować negatywnym wpływem na wydajność programu (Scott Meyers wspomina o bliźniaczym problemie w **Item 34: Prefer lambdas to `std::bind`** w [16]). Skoro czasem *trzeba* skorzystać z `auto`, a – jak się do tej pory wydaje – nie niesie to ze sobą żadnych minusów, czemu by nie korzystać z tego stylu zawsze?

Jednym z argumentów przeciwko adaptacji tego stylu jest teza, że zaciemnia on typ deklarowanej zmiennej. Argument ten jest jednak wadliwy z dwóch powodów:

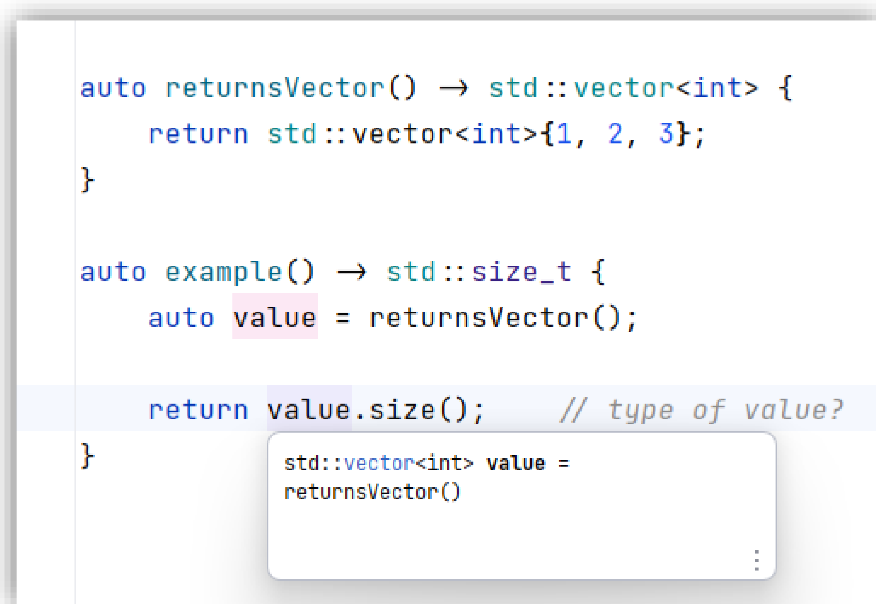
1. W momencie, gdy programistę interesuje konkretny typ (np. chce przeprowadzić jawną konwersję), zawsze może skorzystać ze składni `auto variable = type(init);`, gdzie `type` to typ, który programista chce wymusić, a `init` to wyrażenie inicjalizujące (które w tym przypadku może być puste).

2. Z reguły programisty **wcale nie interesuje, jakiego typu jest dana zmienna**, mimo że może się wydawać inaczej. Doskonałym tego przykładem jest reguła programowania do interfejsu (ang. *Programming to an Interfafce*), która traktuje o tym, że nie jest istotny konkretny typ zmiennej lub parametru, a jedynie zestaw operacji, które oferuje. Przykładowo, w Javie, w większości przypadków zamiast akceptować argument jako `ArrayList<T>` (lub jakąś inną, konkretną klasę) przyjmuje się interfejs `List<T>`, aby zaznaczyć, że implementacja jest drugorzędną kwestią – pod interfejsem, który ta implementacja respektuje.

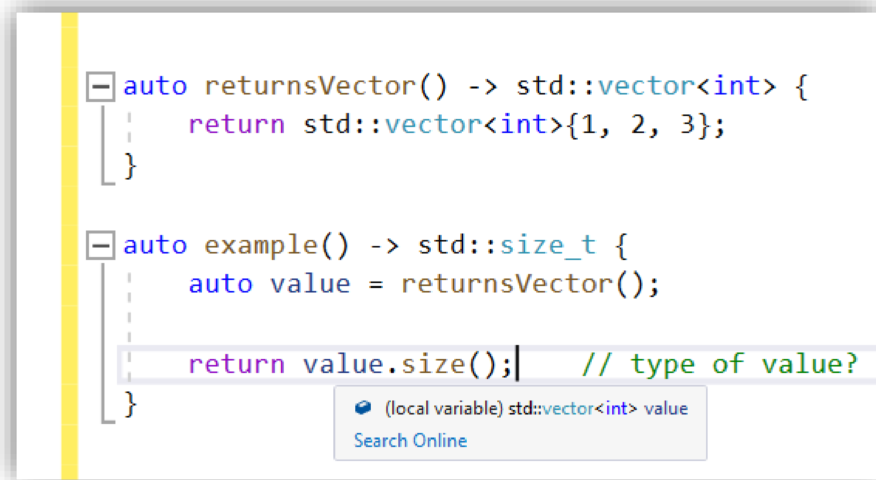
Kolejnym, lecz o nieco mniejszej wadze, kontrargumentem na powyższy zarzut przeciwko AAA, jest stan zaawansowania obecnych narzędzi programistycznych. Przykładowo, środowiska CLion i Visual Studio są bardzo popularnymi i bogatymi w udogodnienia narzędziami wspomagającymi pisanie kodu. Obydwa wspierają narzędzie podręcznej dokumentacji (ang. *quick documentation*), które pozwala na sprawdzenie typu danej zmiennej za pomocą jednego skrótu klawiszowego. To dlatego, że typ jest znany podczas kompilacji, nawet jeżeli się go „ukryje" za deklaracją korzystającą z `auto`. Przykład prezentuje kombinacja Listing 14, Rysunek 1 i Rysunek 2.

Listing 14: Deklaracja zmiennej bez widocznego konkretnego typu (użycie AAA).

```
1  auto returnsVector() -> std::vector<int> {
2      return std::vector<int>{1, 2, 3};
3  }
4
5  auto example() -> std::size_t {
6      auto value = returnsVector();
7
8      return value.size();    // type of value?
9  }
```



Rysunek 1: Okno podręcznej dokumentacji wywołane na rzecz zmiennej `value` w środowisku CLion.



Rysunek 2: Okno podręcznej dokumentacji wywołane na rzecz zmiennej `value` w środowisku Visual Studio.

Jak widać w zaprezentowanym przykładzie, środowiska, dzięki informacjom dostępnym w czasie kompilacji, są w stanie zaspokoić ciekawość użytkownika dotyczącą konkretnego typu danej zmiennej.

Jest to argument o mniejszej wadze, bo jest zależny od preferencji (lub czasem odgórnych wymagań) dotyczących środowisk i narzędzi programistycznych. Niemniej jednak warto zauważyć, że nawet „najmniej bogate" środowiska czy nawet edytory (np. Neovim), za pomocą prostych wtyczek (pluginów) potrafią wspierać dokładnie takie samo zachowanie.

Styl AAA jest też konsekwentny z notacją funkcji precyzującą typ na końcu (ang. *function declaration with trailing return type*) [17] oraz z notacją UML, co niekiedy stanowi ważny argument ku konsekwentności i spójności elementów języka i ekosystemu.

Warto też zwrócić uwagę na fakt, że ogromna liczba (głównie nowych, ale również i niektórych starszych) języków programowania również wspiera (lub zaczyna wspierać) notację, której forma jest zgodna z AAA. Notacja ta respektuje styl polegający na tym, że **po lewej stronie precyzuje się nazwę, a po prawej typ**. Tabela 1 przedstawia wybrane języki i ich składnię deklarowania zmiennych, która respektuje przytoczoną regułę.

Tabela 1: Przykłady deklaracji zmiennych respektujących styl nazwy po lewej stronie i typu po prawej dla wybranych języków.

Język	Deklaracja

Rust	`let count: i32;`
	`let value: f32;`
	`let letter: char;`

Swift	`var count: Int`
	`var value: Float`
	`var letter: Character`

Kotlin	`var count: Int`
	`var value: Float`
	`var letter: Char`

Go	`var count int`
	`var value float32`
	`var letter rune`

Pascal	`var`
	`  count: Integer;`
	`  value: Real;`
	`  letter: Char;`

Ada	`count: Integer;`
	`value: Float;`
	`letter: Character;`

TypeScript	`let count: number;`
	`let value: number;`
	`let letter: string;`

Zig	`var count: i32;`
	`var value: f64;`
	`var letter: u8;`

Nim	`var count: int`
	`var value: float`
	`var letter: char`

Jai	`count: int`
	`value: float`
	`letter: char`

Powyższy przykład (konstrukcje w danych językach przedstawione w Tabela 1) nie jest wyczerpujący, a języki, które są relatywnie wiekowe, to jedynie Pascal i Ada. Można zatem założyć, że trend nowych narzędzi oscyluje wokół stylu konsekwentnego do AAA.

Mniejsza szansa na przypadkową konwersję stratną również jest argumentem ku adaptacji tego stylu, ale została już omówiona dokładniej w [13].

Na koniec warto jeszcze zwrócić uwagę na fakt, że wraz z wejściem standardu C++17, operacja `auto variable = ExpensiveToMoveType(init);` nie powoduje już żadnych problemów z wydajnością. Formalnie stosowanie tego stylu wraz ze sprecyzowaniem typu skutkuje stworzeniem tymczasowej zmiennej, która jest później przenoszona[13] lub kopiowana na potrzeby stworzenia obiektu `variable` (ang. *move-constructed or copy-constructed*). Nie jest to już problemem dlatego, że wraz z tym standardem wprowadzona została gwarancja pomijania kopii (ang. *copy-elision guarantee*) [18]. Aplikuje się to też do niekopiowalnych i nieprzenaszalnych typów, takich jak `std::lock_guard`.

[13] Kontekst dotyczy semantyk przenoszenia.

3.4 Kompletny C++

Język C++, mimo wprowadzenia tak wielu potrzebnych i wygodnych narzędzi, nawet w wersji 17 był „daleko w tyle" w porównaniu do narzędzi dostępnych w innych językach – głownie wyższego poziomu.

Poziom abstrakcji to trudny temat, a klasyfikacja języków według niego budzi wiele kontrowersji. Istnieją argumenty przemawiające ku temu, aby traktować język C jako język wysokiego poziomu z uwagi na fakt, że obrazował on abstrakcję nad kodem maszynowym (lub kodem Assembly), a nie jedynie prostą translację. Z drugiej strony programiści języka Groovy potrafią stwierdzić, że Java jest językiem niskiego poziomu z uwagi na brak wielu istotnych abstrakcji, takich jak dynamiczne typowanie czy przeciążanie operatorów.

Język C++ oferuje narzędzia pasujące do obydwu grup. Z jednej strony wspiera pracę na „surowym metalu" (ang. *bare metal*) za pomocą wskaźników, kompilacji do natywnego kodu maszynowego i wstawek Assembly, a z drugiej oferuje narzędzia wysokiej abstrakcji takie jak szablony, przeciążanie operatorów czy literały zdefiniowane przez użytkownika.

Standard C++20 wprowadził wiele kluczowych, obecnych w innych językach wysokiego poziomu, abstrakcji, które pozwoliły na wzniesienie programowania w nim na wyższy poziom komfortu. Do tych narzędzi należą, między innymi:

- Koncepty [19], [20]
- Zakresy (ang. *Ranges*) [21]
- Moduły (ang. *Modules*) [22], [23]
- Korutyny (ang. *Coroutines*) [24]

Korutyny i Moduły zasługują na osobny, obszerny opis i nie będą omawiane w niniejszej pracy. To dlatego, że same w sobie nie są powiązane z programowaniem generycznym. Moduły służą do prostszego, bezpieczniejszego i wydajniejszego podziału aplikacji C++owych na wiele plików, a korutyny do prostszego projektowania asynchronicznych narzędzi.

Następne rozdziały dokładniej przybliżą temat Konceptów oraz Zakresów. Wpierw zostaną przeanalizowane Koncepty, które są ściśle związane z programowaniem generycznym. Znacznie upraszczają pierwotnie trudny problem ograniczania typów szablonowych (proces w pewien sposób podobny do ograniczania typów generycznych w Javie) oraz pozwalają na klarowniejsze precyzowanie interfejsu. Dodatkowo ich użycie może skutkować klarowniejszymi błędami kompilacji przy używaniu kodu generycznego. Wpierw jednak warto przyjrzeć się samym błędom, z którymi można się spotkać podczas tworzenia generycznych narzędzi.

3.4.1 Błędy zgłaszane przez narzędzia przy używaniu narzędzi generycznych

Dobrej jakości narzędzia programistyczne cechują się weryfikowalnością swojej poprawności bazując na swojej konstrukcji (ang. *correct by construction*). Jeżeli programista może napisać dany kod, to powinien on robić dokładnie to, co programista chce. Przez „może napisać dany kod" rozumie się możliwość napisania fragmentu programu, który się skompiluje, a w przypadku niektórych języków (takich jak Python, z uwagi na dynamiczne typowanie i rozbudowane mechanizmy rozszerzalności obiektów i klas, np. przy pomocy refleksji) skompiluje i uruchomi, wykonując wszystkie opisane kroki.

Weryfikowalność poprawności jest oczywiście czasami niemożliwa. Nie zmienia to jednak faktu, że jeżeli istnieje możliwość automatyzacji takiej weryfikacji (nawet częściowo) to warto rozważyć stworzenie narzędzia, które by to robiło. Przykładowo, na tej zasadzie działa sprawdzanie poprawności dostarczonych typów w językach statycznie typowanych, takich jak Java czy C++. Funkcja lub metoda, która oczekuje pojedynczej zmiennej typu tekstowego, spowoduje błąd kompilacji, jeżeli zostanie podjęta próba wywołania jej z, przykładowo, trzema argumentami typu liczb całkowitych.

Im więcej tego typu weryfikacji można przenieść na barki jakiś narzędzi, tym bardziej produktywna może być praca programisty. Dlatego warto przyjrzeć się mechanizmom weryfikacji oraz szczegółom zgłaszania błędów opisanych wyżej w kontekście programowania generycznego.

Jako przykład posłużą wprowadzone wcześniej listingi kodu przedstawiające sortowanie bąbelkowe w Javie, Pythonie i C++. Zostaną one jednak wzbogacone o ich użycia. Należy przyjrzeć się nie tylko ich implementacji, ale także i błędom, które zgłaszają narzędzia przy próbie niepoprawnego użycia ww. narzędzi.

Listing 15: Przykład użycia kodu generycznego ze wspieranym i niewspieranym typem w Pythonie.

```
 1  def bubble_sort(arr):
 2      n = len(arr)
 3
 4      for i in range(n):
 5          for j in range(0, n - i - 1):
 6              if arr[j] > arr[j + 1]:
 7                  arr[j], arr[j + 1] = arr[j + 1], arr[j]
 9
 8  class Uncomparable:
10      pass
11
12  bubble_sort([3, 2, 1])
13  bubble_sort([Uncomparable()] * 3)
```

Listing 15 przedstawia dwa wywołania Pythonowej funkcji `bubble_sort()` – raz z listą trzech liczb całkowitych, a raz z listą trzech obiektów, których nie da się porównać ze sobą za pomocą operatora >. Błąd, który się pojawia, został pokazany w Listing 16.

Listing 16: Treść błędu traktującego o niewspieranej operacji dla danego typu w Pythonie.

```
1  Traceback (most recent call last):
2    File "/fake/path/to/main.py", line 13, in <module>
3      bubble_sort([Uncomparable()] * 3)
4    File "/fake/path/to/main.py", line 6, in bubble_sort
5      if arr[j] > arr[j + 1]:
6  TypeError: '>' not supported between instances of 'Uncomparable' and 'Uncompara-
   ble'
```

Opisany błąd (czasu wykonania) jest jasny i klarowny. Wskazuje na konkretny, problematyczny fragment kodu (`if arr[j] > arr[j + 1]`) oraz go zwięźle i czytelnie opisuje.

Listing 17: Przykład użycia kodu generycznego ze wspieranym i niewspieranym typem w Javie.

```
1  public class Main {
2      public static <T extends Comparable<? super T>>
3      void bubbleSort(T[] arr) {
4          int n = arr.length;
5
6          for (int i = 0; i < n - 1; i++) {
7              for (int j = 0; j < n - i - 1; j++) {
9                  if (arr[j].compareTo(arr[j + 1]) > 0) {
8                      T temp = arr[j];
10                     arr[j] = arr[j + 1];
11                     arr[j + 1] = temp;
12                 }
13             }
14         }
15     }
16
17     static class Uncomparable { }
18
19     public static void main(String[] args) {
20         bubbleSort(new Integer[]{3, 2, 1});
21         bubbleSort(new Uncomparable[]{
22                 new Uncomparable(),
23                 new Uncomparable(),
24                 new Uncomparable()
25         });
26     }
27 }
```

Listing 17 obrazuje użycie Javowej implementacji `bubbleSort()` z, podobnie jak w przykładzie z językiem Python, typem, który spełnia wymagania bycia sortowalnym i z tym, który ich nie spełnia.

Tutaj również pojawi się błąd, ale jego treść pozostawia nieco więcej do życzenia:

Listing 18: Treść błędu traktującego o niewspieranej operacji dla danego typu w Javie.

```
\fake\path\to\Main.java:21:9
java: method bubbleSort in class Main cannot be applied to given types;
  required: T[]
  found:    Main.Uncomparable[]
  reason: inference variable T has incompatible bounds
    lower bounds: java.lang.Comparable<? super T>
    lower bounds: Main.Uncomparable
```

Jak można zauważyć w Listing 18, kompilator Javy poprawnie diagnozuje problem z przekazanym typem, a konkretniej z niespełnieniem wymagań przez ten typ. Typ `T` musi być porównywalny z typem `T` lub z jakimkolwiek typem „wyżej" w tej hierarchii dziedziczenia (notacja `? super T`).

Oryginalnie może się to wydawać nieco nieintuicyjne, ale prosty przykład może zobrazować poprawność tej logiki: Załóżmy istnienie typu, który jest ze sobą porównywalny, np. `Drink`. Klasa `Drink` implementuje interfejs `Comparable<Drink>`, dzięki czemu można porównać dwa obiekty tego typu. Teraz załóżmy istnienie dwóch klas pochodnych – `Whisky` oraz `Absinthe`. Te dwa typy nie implementują żadnego dodatkowego interfejsu, ale – dzięki dziedziczeniu – efektywnie implementują `Comparable<Drink>`. Na koniec załóżmy istnienie tablicy `Drink[]`, która przechowuje obiekty typów dziedziczących (`Whisky` oraz `Absinthe`). Obiekt `Whisky` można porównać z drugim obiektem typu `Whisky`, ponieważ ich metoda `compareTo()` (należąca do interfejsu `Comparable`, którą odziedziczyli) przyjmuje `Drink`, czyli ich klasę bazową. To samo z `Absinthe`. Dodatkowo sprawia to, że można porównać ze sobą obiekty typów `Absinthe` i `Whisky`. Zakładając, że mamy do czynienia wyłącznie z tablicą typu `Drink`, ograniczenie `? super T` nie byłoby potrzebne, ale daje ono dodatkowy plus. Gdyby stworzyć tablicę typu `Whisky` (a nie klasy bazowej `Drink`), to taka tablica przechowywała elementy, które nie są porównywalne ze sobą per se. Są porównywalne z jakąś swoją klasą bazową (tutaj – `Drink`). Metoda `compareTo()` wciąż by działała, ale już `T extends Comparable<T>` dla `Whisky` nie byłoby spełnione. Metoda `bubbleSort()` wtedy nie mogłaby przyjąć obiektu `Whisky[]`, mimo tego, że można wywołać na ich obiektach metodę `compareTo()`.

Efektywnie, błąd z Listing 18 mówi, że „`bubbleSort()` potrzebuje tablicy obiektów spełniających dane wymagania (gdzie te wymagania to implementacja `Comparable` na jakimś poziomie dziedziczenia), ale otrzymała tablicę obiektów, które ich nie spełniają." Błąd nieco mniej jasny niż w przypadku Pythona.

W przypadku języka C++ błąd wskazuje bezpośrednio na problematyczny `operator >`, choć kontekst nieco przyćmiewa ten fakt.

Listing 19: Przykład użycia kodu generycznego ze wspieranym i niewspieranym typem w C++.

```cpp
#include <vector>
#include <utility>  // swap

template <typename T>
auto bubbleSort(std::vector<T>& vec) -> void {
    auto n = vec.size();
```

```
 7
 9      for (auto i = 0; i < n - 1; i++) {
 8          for (auto j = 0; j < n - i - 1; j++) {
10              if (vec[j] > vec[j + 1]) {
11                  std::swap(vec[j], vec[j + 1]);
12              }
13          }
14      }
15  }
16
17  class Uncomparable { };
18
19  auto main() -> int {
20      auto numbers       = std::vector<int>{3, 2, 1};
21      auto uncomparables = std::vector<Uncomparable>{
22              {}, {}, {}
23      };
24      bubbleSort(numbers);
25      bubbleSort(uncomparables);
26  }
```

Dla kompletności zaprezentowane zostaną błędy z trzech wiodących kompilatorów języka C++ dla kodu z Listing 19.

Listing 20: Treść błędu traktującego o niewspieranej operacji dla danego typu w C++ wyprodukowana przez kompilator GCC 13.1.

```
1  <source>: In instantiation of 'void bubbleSort(std::vector<T>&) [with T = Uncompa-
   rable]':
2  <source>:25:15:   required from here
3  <source>:10:24: error: no match for 'operator>' (operand types are
   '__gnu_cxx::__alloc_traits<std::allocator<Uncomparable>,
   Uncomparable>::value_type' {aka 'Uncomparable'} and
   '__gnu_cxx::__alloc_traits<std::allocator<Uncomparable>,
   Uncomparable>::value_type' {aka 'Uncomparable'})
4     10 |             if (vec[j] > vec[j + 1]) {
5        |                 ~~~~~~~^~~~~~
6  ASM generation compiler returned: 1
```

Listing 21: Treść błędu traktującego o niewspieranej operacji dla danego typu w C++ wyprodukowana przez kompilator Clang 16.0.

```
1  <source>:10:24: error: invalid operands to binary expression ('value_type' (aka
   'Uncomparable') and 'value_type')
2              if (vec[j] > vec[j + 1]) {
3                  ~~~~~~ ^ ~~~~~~~~~~
4  <source>:25:5: note: in instantiation of function template specialization
   'bubbleSort<Uncomparable>' requested here
5      bubbleSort(uncomparables);
6      ^
7  1 error generated.
```

```
9 Execution build compiler returned: 1
```

Listing 22: Treść błędu traktującego o niewspieranej operacji dla danego typu w C++ wyprodukowana przez kompilator MSVC.

```
1 <source>(10): error C2676: binary '>': '_Ty' does not define this operator or a
  conversion to a type acceptable to the predefined operator
2         with
3         [
4             _Ty=Uncomparable
5         ]
6 <source>(25): note: see reference to function template instantiation 'void
  bubbleSort<Uncomparable>(std::vector<Uncomparable,std::allocator<Uncomparable>>
  &)' being compiled
  Compiler returned: 2
7
```

Mimo większych lub mniejszych różnic, dosyć jasno widać, że jakość błędów zgłaszanych przez kompilatory C++ pozostawia wiele do życzenia. Zwłaszcza w sytuacji, gdzie istnieje kilka szablonów będących kandydatami do użycia w danym wywołaniu. Problemy te były znane komitetowi standaryzacyjnemu języka C++ (grupa ISO WG21). Projekt rozwiązania również był znany. Dotyczył on Konceptów.

3.4.2 Koncepty

Koncept to byt pozwalający na precyzowanie zestawu funkcjonalności. Konkretne typy mogą dany Koncept spełniać lub go nie spełniać. Formalnie nazywa się je *nazwanymi zestawami wymagań / ograniczeń* (ang. *named set of constraints*). Gdy projektuje się kod generyczny, mogą pojawić się dwa problemy, z którymi to narzędzie może programiście pomóc:

1. Chęć zakomunikowania, z jakim rodzajem typów generycznych dany kod może współpracować.
2. Chęć dostarczenia zestawu generycznych narzędzi (zamiast pojedynczego generycznego narzędzia) pracujących na różnych typach, korzystających z różnych implementacji (nielegalnych dla niektórych typów).

Warto przyjrzeć się przykładom obydwu tych motywacji, wpierw zaczynając od drugiego z nich.

3.4.2.1 Tworzenie zestawu generycznych narzędzi, które akceptują grupę typów

Załóżmy generyczną funkcję (szablon funkcji) wyświetlającą przekazany argument, zaprezentowaną w Listing 23.

Listing 23: Szablon funkcji `print()`.

```
1  template <typename T>
2  auto print(T const& t) -> void {
3      std::cout << t << '\n';
4  }
```

Szablon funkcji `print()` z Listing 23 pozwala na wywołanie go z dowolnym argumentem. Jeżeli argument można wyświetlić (czyli wspiera `operator<<` z lewym operandem będącym typu `std::ostream&`), to kompilacja przebiegnie pomyślnie i uruchomienie programu poskutkuje wyświetleniem przekazanego obiektu. Jeżeli jednak nie będzie to możliwe, program nie przejdzie fazy kompilacji i wyprodukuje błąd informujący o tym, że dany typ `T` nie może być prawym operandem dla operatora `<<`.

W języku C++ kolekcje takie jak `std::vector`, `std::map` czy `std::queue` nie wspierają operacji wyświetlania. Oznacza to, że gdy prześlemy obiekty tych typów do omawianego szablonu `print()`, program nie przejdzie kompilacji z powodów przytoczonych wyżej.

Sposób naprawy tego problemu może się wydawać trywialny – należy zwyczajnie dodać przeciążenie szablonu `print()`, które będzie akceptował dowolny typ i, traktując go jako jakiś zakres, wyświetlał jego elementy. Listing 24 przedstawia przykład takiego rozwiązania wraz z przykładami użycia.

Listing 24: Próba przeciążenia szablonu `print()` dla „zwykłych" obiektów oraz dla zakresów.

```
1  template <typename T>
2  auto print(T const& t) -> void {
3      std::cout << t << '\n';
4  }
5
6  template <typename Container>
7  auto print(Container const& container) -> void {
8      for (auto const& element : container) {
9          std::cout << element << ' ';
10     }
11     std::cout << '\n'
12 }
13
14 auto main() -> int {
15     auto singleNumber = 123;
16     auto numbers      = std::vector<int>{1, 2, 3};
17     print(singleNumber);
18     print(numbers);
19 }
```

Niestety, o ile pomysł wygląda dobrze, powyższy kod się nie skompiluje. C++ traktuje obydwie definicje `print()` jako tożsame. `<typename T>` oraz `<typename Container>` opisują jeden argument szablonowy, którego nazwa nie jest istotna dla kompilatora. Redefinicja funkcji z identyczną sygnaturą jest nielegalna. Błąd, który zgłasza kompilator GCC 12.2.0 kompilujący ten kod zgodnie ze standardem C++20 ma **340 linijek długości**. Większość z nich traktuje o potencjalnych (ale niekompatybilnych)

kandydatach w postaci operatorów `<<` (z pierwszej implementacji `print()`). Na szczęście błąd zawiera też przydatne fragmenty, takie jak te zaprezentowane w Listing 25.

Listing 25: Fragment błędu wygenerowanego na skutek niepoprawnej redefinicji szablonu `print()`.

```
1  main.cpp:10:6: error: redefinition of 'template<class Container> void print(const
   Container&)'
2     10 | auto print(Container const& container) -> void {
3        |      ^~~~~
4  main.cpp:5:6: note: 'template<class T> void print(const T&)' previously declared
   here
5      5 | auto print(T const& t) -> void {
6        |      ^~~~~
7  main.cpp: In instantiation of 'void print(const T&) [with T = std::vector<int>]':
   main.cpp:21:10:   required from here
8  main.cpp:6:15: error: no match for 'operator<<' (operand types are 'std::ostream'
9  {aka 'std::basic_ostream<char>'} and 'const std::vector<int>')
       6 |     std::cout << t;
         |     ~~~~~~~~~~^~~~
10
11
```

Pomysł dostarczenia dwóch generycznych narzędzi osiągających ten sam cel, ale wspierających różne grupy typów, nie jest zły sam w sobie. Język C++ od momentu wprowadzenia szablonów wspierał mechanizm pozwalający ograniczyć te typy. Niestety nie wprost. Mechanizm, o którym mowa, nazywa się SFINAE (ang. *Substitution Failure Is Not An Error*). Traktuje o tym, że niepowodzenie podstawiania typu (w kontekście dedukcji typu szablonowego) nie jest (nie musi być) błędem.

SFINAE polega na wprowadzeniu dodatkowego punktu dedukcyjnego dany typ. Dedukcja nie powiedzie się, jeżeli wspomniany typ nie będzie spełniał danych wymogów. Przykładem ograniczenia drugiego przeciążenia szablonu `print()` mógłby być wymóg, aby typ `Container` był faktycznie kontenerem lub jakimkolwiek zakresem elementów.

Język C++ niestety nie posiada sprecyzowanego zestawu cech, które określają, czy dany typ jest kontenerem / kolekcją. Istnieje jednak definicja zakresu danych, która – nieco uproszona – traktuje o tym, że na danym typie musi się dać wywołać metody `.begin()` oraz `.end()`, które zwracają iteratory.

Jak zatem ograniczyć drugie przeciążenie `print()` za pomocą SFINAE tak, aby akceptowało tylko typy, które są zakresami elementów? Jak upewnić się, że gdy typ będzie zakresem, to *tylko* drugie przeciążenie będzie wybrane, zamiast pasować do obydwu?

Mechanik SFINAE jest wiele. Najprostszym rozwiązaniem powyższego problemu jest skorzystanie z wyrażenia SFINAE (ang. *expression SFINAE*). Narzędziem, które pozwoli na skorzystanie z tej strategii jest *trailing return type*, który został przytoczony w rozdziale **3.3.1**. Jest to przykład demonstrujący przydatność tego narzędzia, co jest bezpośrednio kolejnym argumentem przemawiającym ku adaptacji AAA. Warto tutaj jeszcze przypomnieć, że jest to narzędzie wprowadzone w standardzie C++11. W starszych standardach trzeba byłoby skorzystać ze znacznie trudniejszego w użyciu metaprogramowania bazującego na rozwiązaniach emulujących typ `std::enable_if`.

Listing 26: Przykład użycia SFINAE bazującego na funkcji z określonym typem na końcu.

```cpp
#include <iostream>
#include <vector>
#include <type_traits>  // void_t

namespace details {
    template <typename Container>
    auto print(
            Container const& container, int
    ) -> std::void_t<decltype(container.begin(), container.end())> {
        for (auto const& element : container) {
            std::cout << element << ' ';
        }
    }

    template <typename T>
    auto print(T const& t, double) -> void {
        std::cout << t << '\n';
    }
}

template <typename T>
auto print(T const& t) -> void {
    details::print(t, 0);
}

auto main() -> int {
    auto singleNumber = 123;
    auto numbers      = std::vector<int>{1, 2, 3};
    print(singleNumber);
    print(numbers);
}
```

Składnia `auto nazwaFunkcji(argumenty) -> TypZwracany` w połączeniu ze słowem kluczowym `decltype()` w sposób zaprezentowany w Listing 26 pozwala na stworzenie sygnatury szablonu funkcji, który będzie brany pod uwagę tylko w przypadku, gdy sprecyzowane operacje będą wspierane. Jeżeli operacje te nie będą wspierane, wybrany zostanie inny szablon z pasującą sygnaturą.

W tym przykładzie szablon `print()` jest wywoływany dwa razy – wpierw dla argumentu typu `int`. Mając do dyspozycji dwóch kandydatów `details::print()`, kompilator sprawdza, który z nich pasuje do danego wywołania. Pierwszy (linijka 7) nie pasuje. To dlatego, że o ile argumenty można dopasować, to typ zwracany zależy od wyrażenia `container.begin(), container.end()`, gdzie `container` byłby tym przekazanym `intem`. Na zmiennej tego typu nie można wywołać tych metod (na `intach` nie można wywołać żadnych metod), a zatem mechanizm SFINAE **dyskwalifikuje to przeciążenie**. Może się to wydawać dziwne, że typ zwracany zależy od tego, czy możliwym jest wywołanie dwóch wspomnianych metod. W końcu `void_t` zawsze sprowadza się do reprezentowania typu `void`. Niemniej jednak kompilator musi sprawdzić poprawność wyrażenia użytego w argumencie szablonu `void_t`. Wyrażenie jest niepoprawne z powodu niewspieranych operacji. W pewnym sensie jest to oszukana zależność, ale z uwagi na zasady działania języka, osiąga swój cel.

Drugi kandydat (linijka 16) jest dopasowywany bez problemu.

W przypadku drugiego wywołania (wykorzystującego `std::vector`) sytuacja jest podobna. Kompilator na początek sprawdza, który z przeciążeń pasuje do wywołania. Odpowiedź na to pytanie może być zaskakująca, bo **pasują obydwa**. W końcu wysłanie obiektu typu `std::vector` oraz zmiennej typu `int` (literał `0` z linijki 23) pasuje zarówno do sygnatury `template <typename Container> auto print(Container const& container, int) -> std::void_t<decltype(container.begin(), container.end())>` jak i do sygnatury `template <typename T> auto print(T const& t, double) -> void`.

Jaki mechanizm zatem sprawia, że powyższy kod poprawnie przechodzi proces kompilacji?

Odpowiedzialny jest za to drugi argument szablonów `details::print()`. Jeżeli dany typ pasuje do obydwu szablonów, należy mieć możliwość sprecyzowania, który z nich powinien być „domyślny". Domyślnym powinien być ten „bardziej precyzyjny", który dokładniej pasuje do typu. Dlatego ten domyślny szablon jest opatrzony dodatkowym, nieużywanym parametrem typu `int`, gdzie drugi z nich jest opatrzony dodatkowym, nieużywanym parametrem typu `double`. Gdy wywoła się `details::print()` z argumentem będącym literałem `0`, to obydwa przeciążenia będą legalnymi kandydatami, ale ten akceptujący `int` będzie **lepszym** kandydatem od tego, który akceptuje `double`. To dlatego, że język C++ definiuje priorytet kandydatów, który zależy od, miedzy innymi, liczby wymaganych niejawnych konwersji.

Mając funkcję, która przyjmuje parametr typu `double`, można ją wywołać przesyłając argument typu `int`. Zajdzie niejawna konwersja. Ale gdy doda się przeciążenie przyjmujące typ `int`, to będzie ono lepszym kandydatem przy wyborze przeciążeń, bo nie wymaga dodatkowej konwersji. Dokładnie ten mechanizm został wykorzystany w powyższym przykładzie.

Nie sposób jednak nie zauważyć, że przynajmniej część z tych narzędzi i mechanizmów bardziej przypomina obejścia (ang. *workarounds* / *hacks*) niż użycie narzędzi zgodnie z ich przeznaczeniem. Dodatkowo używanie drugiego, niewykorzystywanego parametru w celu sprecyzowania hierarchii precyzyjności jest rozwiązaniem nieskalowalnym. Co gdyby chcieć stworzyć nie dwa, a – przykładowo – pięć przeciążeń z różnymi wariantami? Takie ubogie uproszczenie mechanizmu ekspedycji typów (ang. *tag dispatch*) [25] [26] nie sprawia wrażenia bycia odpowiednim narzędziem.

Kod z Listing 26 jest zgodny ze standardem C++11. Czy w standardzie C++14 lub C++17 dodano jakieś narzędzia, które mogłyby go uprościć?

W pewnym sensie tak, w pewnym sensie nie. Najpewniej czytelniejszym rozwiązaniem byłoby skorzystanie z `if constexpr()`, ale i tak trzeba by korzystać z zaawansowanych technik metaprogramowania do sprawdzenia, czy dany typ jest zakresem. Wynikowy kod byłby równie skomplikowany. Sama implementacja `print()` byłaby dużo prostsza, ale wymagane do stworzenia pomocnicze narzędzia byłyby nieco bardziej skomplikowane.

Na szczęście w C++20 wprowadzono Koncepty. Dzięki nim można osiągnąć ten sam efekt (stworzyć `print()` odpowiednio działający dla „zwykłych" obiektów jak i dla zakresów) we wręcz trywialny, zaprezentowany w Listing 27, sposób.

Listing 27: Przykład użycia Konceptów.

```
#include <iostream>
#include <vector>
#include <ranges>   // range

template <typename T>
auto print(T const& t) -> void {
    if constexpr (std::ranges::range<T>) {
        for (auto const& element : t) {
            std::cout << element << ' ';
        }
    } else {
        std::cout << t;
    }
    std::cout << '\n';
}

auto main() -> int {
    auto singleNumber = 123;
    auto numbers      = std::vector<int>{1, 2, 3};
    print(singleNumber);
    print(numbers);
}
```

`if constexpr()` [27] [28] (C++17), bazując na logicznej wartości znanej w czasie kompilacji, gdy jest częścią szablonu, pozwala na warunkowe generowanie (a zatem i kompilowanie) kodu. W Listing 27 jest on użyty wraz z Konceptem `std::ranges::range`, który dla danego typu ewaluuje do wartości `true`, jeżeli spełnia on ten Koncept[14]. W takim przypadku nie zostanie wygenerowany kod z gałęzi `else`, dzięki czemu kompilator nie będzie musiał raportować błędu dotyczącego faktu, że danego zakresu nie da się wyświetlić używając `<<` (jak w Listing 25).

Aby dokładniej zaprezentować możliwości Konceptów, warto przyjrzeć się implementacji tego samego kodu, ale zamiast używania `if constexpr()` zostaną użyte (znane już wcześniej) przeciążenia szablonów. Zamiast pojedynczego `print()` z Listing 27 można skorzystać z przeciążeń przedstawionych w Listing 28.

Listing 28: Przykład prezentujący jak Koncepty poprawnie współpracują z przeciążaniem szablonów.

```
template <std::ranges::range T>
auto print(T const& t) -> void {
    for (auto const& element : t) {
        std::cout << element << ' ';
    }
    std::cout << '\n';
}
```

[14] Dany typ spełnia Koncept `std::ranges::range` jeżeli można wywołać `std::ranges::begin()` oraz `std::ranges::end()` na jego obiektach. Z kolei możliwość wywołania tych dwóch funkcji zależy od tego, czy istnieje metoda (lub wolna funkcja o tej samej nazwie), którą można wywołać na danym typie, która z kolei zwróci obiekt, którego typ spełnia Koncept odpowiedniego iteratora.

```
 8
 9  template <typename T>
10  auto print(T const& t) -> void {
11      std::cout << t << '\n';
12  }
```

Należy zwrócić uwagę na dwie istotne rzeczy:

- W pierwszej linii zamiast znanej składni `template <typename T>` widać `template <std::ranges::range T>`. Oznacza to, że generyczny typ `T` nie reprezentuje *dowolnego* typu, a jedynie taki, który *spełnia Koncept* `std::ranges::range`. Co ciekawe, te składnię stosował już od dawna Alexander Stepanov, autor podwalin standardowej biblioteki szablonowej (STL) C++ [29]. Składnię te można zaobserwować w obszernych przykładach w jego książce, która traktuje o połączeniu matematyki i programowania generycznego [30].
- Mimo tego, że obiekt typu `std::vector` pasowałby do obydwu przeciążeń, to wysyłając go jako argument, wybrane zostanie pierwsze przeciążenie (korzystające z Konceptu). Dzieje się tak z uwagi na zasadę traktującą o tym, że gdy szablon jest *dokładniej dopasowany*, jest też lepszym kandydatem. `typename T` pasuje do wszystkiego. `std::ranges::range T` tylko do zakresów. Dopasowanie do węższego grona typów jest traktowane jako **dokładniejsze dopasowanie**.

Wraz z wprowadzeniem Konceptów, funkcjonalność słowa kluczowego `auto` została rozszerzona. Listing 28 można zapisać alternatywnie stosując składnię zaprezentowaną w Listing 29.

Listing 29: Przykład alternatywnej składni definiowania argumentu generycznego.

```
 1  auto print(std::ranges::range auto const& t) -> void {
 2      for (auto const& element : t) {
 3          std::cout << element << ' ';
 4      }
 5      std::cout << '\n';
 6  }
 7
 8  auto print(auto const& t) -> void {
 9      std::cout << t << '\n';
10  }
```

Mechanizm ten nazywa się *skróconym szablonem funkcji* [31], *skróconym* `auto` lub *ograniczonym* `auto` (ang. *constrained* `auto`).

Jak widać, składnia wprowadzona wraz z Konceptami pozwala na znacznie prostsze tworzenie zestawów generycznych narzędzi pracujących z podgrupami wszystkich typów.

3.4.2.2 Tworzenie własnych Konceptów i ograniczeń dla typów generycznych

Może się zdarzyć potrzeba zdefiniowania własnego konceptu. Na przykład gdy użytkownik potrzebuje stworzyć szablon, który przyjmie dowolną kolekcję i dowolną liczbę argumentów, które będą dodawane

do tej kolekcji. Stworzenie nazwanego zestawu wymagań pozwoli na klarowniejsze zaprojektowanie interfejsu precyzującego, z jakiego rodzajem typów generycznych dany kod może współpracować (motywacja 1 z początku podrozdziału **3.4.2**).

Załóżmy, że dla ułatwienia rozpatrujemy jedynie kontenery, do których można dodać element za pomocą metody `.push_back()` (np. `std::vector`, `std::string`, `std::deque`) lub `.insert()` (np. `std::set`). Można osiągnąć ten cel tworząc własne Koncepty, dzięki czemu implementacja w połączeniu z `if constexpr()` może wybierać, którą metodę należy użyć. Przykład został zaprezentowany w Listing 30.

Listing 30: Przykład Konceptów zdefiniowanych przez użytkownika.

```
1  #include <concepts> // convertible_to
2  #include <ranges>   // range_value_t
3
4  using std::ranges::range_value_t;
5
6  template <typename Container>
7  concept SupportsPushBack = requires(
8          Container& container,
9          range_value_t<Container> const& newValue
10 ) {
11     container.push_back(newValue);
12 };
13
14 template <typename Container>
15 concept SupportsInsertFor = requires(
16         Container& container,
17         range_value_t<Container> const& newValue
18 ) {
19     container.insert(newValue);
20 };
21
22 using std::convertible_to;
23
24 template <typename Container>
25 auto addAllTo(
26         Container& container,
27         convertible_to<range_value_t<Container>> auto const&... toAdd
28 ) -> void {
29     if constexpr (SupportsPushBack<Container>) {
30         (container.push_back(toAdd), ...);
31     } else {
32         (container.insert(toAdd), ...);
33     }
34 }
```

Przykład użycia kodu z Listing 30 został zaprezentowany w Listing 31.

Listing 31: Przykład użycia szablonu `addAllTo()`.

```
1  auto main() -> int {
2      auto set    = std::set<int>();
3      auto vector = std::vector<int>();
```

```
4
5       addAllTo(set, 1, 2, 3);
6       addAllTo(vector, 4, 5);
7   }
```

Po zaprezentowanym wywołaniu `addAllTo()` dwa razy, obiekt `set` przechowuje trzy elementy, a obiekt `vector` dwa (kolejno `{1, 2, 3}` oraz `[4, 5]`).

Analiza tego szablonu jest relatywnie prosta:

1. `addAllTo()` należy wywołać z dowolnym typem podanym jako pierwszy argument (deklaracja parametru szablonowego `Container` oraz parametru `Container& container`). Najlepiej byłoby sprecyzować Koncept dla kolekcji, ale to nieproporcjonalnie trudne zadanie.

2. Jako drugi parametr dla `addAllTo()` przyjmuje się dowolną liczbę (składnia trzech kropek – `...`) argumentów dowolnego typu (ograniczone `auto`), które są konwertowalne (`std::convertible_to`) do typu elementu przechowywanego przez typ pierwszego argumentu (`std::ranges::range_value_t<Container>`). Dzięki temu można wywołać ten szablon nie tylko dodając kilka `int`ów do kontenera `std::vector<int>`, ale również dodając je do kontenera `std::vector<double>`, w przypadku którego typy dodawanych elementów nie są takie same, jak typ elementów samej kolekcji. Są jednak do nich konwertowalne, a zatem taka operacja powinna być akceptowalna.

3. `if constexpr()` sprawdza, czy dana kolekcja wspiera operację `.push_back()`. Jeżeli tak, to zostanie ona użyta. Jeżeli nie, to zostanie użyta metoda `.insert()`. Dodatkowo, biorąc pod uwagę sposób działania operacji `if()`, jeżeli dany typ wspiera obydwie operacje, pierwsza z nich (`.push_back()` – sprawdzany wcześniej) zostanie użyta.

Składnia definicji Konceptu składa się z, między innymi, jego nazwy oraz z wyrażenia `requires()`, w której użytkownik definiuje operacje, które dany typ musi wspierać, aby dany Koncept spełniać.

`requires()` może być też użyte w środku implementacji (jako wyrażenie) oraz przy nagłówku szablonu (jako klauzula), pozwalając zdefiniować dodatkowe wymogi dla typów (np. wymóg spełniania wielu Konceptów). Wyrażenia `requires()` mogą też być zdefiniowane lokalnie. Kod z Listing 30 może zostać zastąpiony tym z Listing 32.

Listing 32: Implementacja szablonu `addAllTo()` za pomocą lokalnego warunku bazującego na wyrażeniu `requires()`.

```
1   using std::ranges::range_value_t;
2   using std::convertible_to;
3
4   template <typename Container>
5   auto addAllTo(
6           Container& container,
7           convertible_to<range_value_t<Container>> auto const&... toAdd
8   ) -> void {
```

```
 9      using ValueType = range_value_t<Container>;
10
11      constexpr auto supportsPushBack = requires {
12          container.push_back(std::declval<ValueType>());
13      };
14
15      if constexpr (supportsPushBack) {
16          (container.push_back(toAdd), ...);
17      } else {
18          (container.insert(toAdd), ...);
19      }
20  }
```

Plusem takiego rozwiązania jest możliwość używania nazw lokalnych zmiennych (włącznie z parametrami) w wyrażeniu `requires()`, co można zauważyć w linijce 11[15]. Wyrażenie `requires()` zwraca zmienną logiczną – `bool`. Deklaracja tej wartości jako `constexpr` (dostępna w czasie kompilacji) sprawia, że może być użyta w `if constexpr()`, które z kolei – jak zostało to już wcześniej wspomniane – poskutkuje warunkową kompilacją tylko tego kodu, który użytkownik w tym momencie chce, aby był legalny.

Jak widać, narzędzia pozwalające tworzyć ograniczenia dla typów generycznych wprowadzone w standardzie C++20 znacznie ułatwiają tworzenie komponentów generycznych. Są bardzo elastyczne, dzięki czemu można w bardzo wygodny sposób definiować zarówno generyczne (ogólne) jak i lokalne ograniczenia dla szablonów.

3.4.2.3 Reprezentowanie ograniczeń typów generycznych w celu zwiększenia klarowności interfejsu

Warto jeszcze przyjrzeć się przykładowi ostatniego miejsca, gdzie można użyć słowa kluczowego `requires()`. Tuż po liście parametrów szablonowych oraz tuż po typie zwracanym. Tym razem za przykład posłuży fragment implementacji `std::ranges::sort()` z libstdc++ [32] (z nieco zmodyfikowanym formatowaniem w celu zwiększenia czytelności).

Listing 33: Sygnatura i definicja fragmentu biblioteki standardowej odpowiedzialnego za działanie algorytmu `std::ranges::sort()`.

```
1  template<
2          random_access_range Range,
3          typename Comp = ranges::less,
4          typename Proj = identity>
5  requires sortable<iterator_t<Range>, Comp, Proj>
6  constexpr auto operator()(
```

[15] Użyty w tej linijce szablon `std::declval()` służy do manifestacji referencji do sprecyzowanego typu. Został on użyty dlatego, że gdyby umieścić tam wyrażenie `ValueType()` w celu manifestacji obiektu, to wyrażenie to by dodatkowo testowało, czy typ elementów przechowywanych przez kontener jest konstruowalny bezargumentowo (ang. *default-constructible*). Dodatkowo, mogłoby to przypadkiem testować, czy istnieje przeciążenie `.push_back()` dla r-wartości (ang. *rvalues*) zamiast lwartości (ang. *lvalues*), z którymi mamy do czynienia w samej implementacji.

```
 7          Range&& r,
 8          Comp comp = {},
 9          Proj proj = {}
10  ) const -> borrowed_iterator_t<Range> {
11      return (*this)(
12              ranges::begin(r), ranges::end(r),
13              std::move(comp), std::move(proj)
14      );
15  }
```

Patrząc na tak zdefiniowane narzędzie do sortowania, od razu można wyciągnąć bardzo dużo informacji z samej deklaracji. Szablon wymaga zakresu, który udostępnia dostęp do dowolnego elementu w czasie stałym (ang. *random access*) (`Range`). Przyjmuje też dwa parametry z domyślną wartością – komparator oraz projekcję.

Na tych trzech parametrach postawione jest dodatkowo jeszcze jedno ograniczenie. Linijka 5 korzysta z klauzuli `requires` ze sprecyzowanym Konceptem `sortable`. Jego definicja została przedstawiona w Listing 34.

Listing 34: Definicja Konceptu `sortable`.

```
1  template <
2          class I,
3          class Comp = ranges::less,
4          class Proj = std::identity
5  >
6  concept sortable = std::permutable<I> and
7      std::indirect_strict_weak_order<Comp, std::projected<I, Proj>>;
```

Można zauważyć tutaj popularną tendencję przy Konceptach. Jeden korzysta (lub rozszerza) z drugiego. Coś, co można posortować, musi być permutowalne oraz dany komparator, którym użytkownik chce coś posortować, musi spełniać relację słabego uporządkowania (w uproszczeniu – ma intuicyjnie i poprawnie, czyli zgodnie z matematycznymi założeniami, reprezentować relację zdefiniowaną przez operator mniejszości).

Prostota tworzenia, rozszerzania i korzystania z Konceptów sprawia, że kod generyczny pisany w standardzie C++20 (i później) jest dużo bardziej elegancki i prostszy w użyciu w porównaniu do wcześniejszych iteracji języka.

Należy w tym miejscu zauważyć podobieństwo między Konceptami, a ograniczeniami typów generycznych w języku Java, które zostały wprowadzone w rozdziale **2.4.2**. Obydwa te narzędzia pozwalają na sprecyzowanie cech podgrupy typów, do których dany parametr generyczny może się dopasować. Istnieje jednak bardzo ważne rozróżnienie, o którym należy pamiętać. W języku Java, to typ (klasa) „decyduje” o tym, jaki `interface` implementuje i tym samym w jakich kontekstach generycznych może zostać użyty. W języku C++, dany typ (czy to klasa czy typ wbudowany) nie ma świadomości istnienia jakiegokolwiek Konceptu. To Koncept sprawdza, czy dany typ go spełnia, a nie typ zaznacza, że spełnia dany Koncept.

Mając to podobieństwo na uwadze, warto jeszcze raz porównać błędy generowane przez generyczny kod Javy (Listing 18) i C++ (Listing 19). Koncepty pozwalają dokładnie zweryfikować, które **nazwane** wymaganie zostało niespełnione. Jak zatem będzie wyglądał błąd z przykładu zaprezentowanego w Listing 19, gdy wzbogaci się „surowy" parametr szablonowy ograniczeniami w postaci Konceptów? Listing 35 prezentuje zmodyfikowany nagłówek szablonu `bubbleSort()`, a Listing 36 wygenerowany błąd.

Listing 35: Przykład użycia kodu generycznego ograniczonego Konceptami ze wspieranym i niewspieranym typem w C++.

```
1  #include <vector>
2  #include <utility>      // swap
3  #include <ranges>       // std::ranges::*
4  #include <iterator>     // std::sortable
5  #include <functional>   // std::identity
6  template <std::ranges::random_access_range T>
7  requires std::sortable<
8          std::ranges::iterator_t<T>,
9          std::ranges::less, std::identity
10 >
11 auto bubbleSort(T& container) -> void {
12     auto n = container.size();
13     for (auto i = 0; i < n - 1; i++) {
14         for (auto j = 0; j < n - i - 1; j++) {
15             if (container[j] > container[j + 1]) {
16                 std::swap(container[j], container[j + 1]);
17             }
18         }
19     }
20 }
21
22 class Uncomparable { };
23
24 auto main() -> int {
25     auto numbers       = std::vector<int>{3, 2, 1};
26     auto uncomparables = std::vector<Uncomparable>{
27             {}, {}, {}
28     };
29     bubbleSort(numbers);
30     bubbleSort(uncomparables);
31 }
```

Listing 36: Treść błędu kompilacji wygenerowana przez kompilator GCC 12.2.0 dla kodu z Listing 35.

```
1  main.cpp: In function 'int main()':
2  main.cpp:31:15: error: no matching function for call to 'bubbleSort(std::vec-
3  tor<Uncomparable>&)'
4     31 |     bubbleSort(uncomparables);
5        |     ~~~~~~~~~~^~~~~~~~~~~~~~~
6  main.cpp:11:6: note: candidate: 'template<class T>  requires (random_ac-
7  cess_range<T>) &&
```

```
 8  (sortable<decltype(std::ranges::__cust_access::__begin((declval<_Container&>)())),
 9  std::ranges::less, std::identity>) void bubbleSort(T&)'
10     11 | auto bubbleSort(T& container) -> void {
11        |      ^~~~~~~~~~
12  main.cpp:11:6: note:   template argument deduction/substitution failed:
13  main.cpp:11:6: note: constraints not satisfied
14  In file included from c:\mingw64\include\c++\12.2.0\compare:39,
15                   from c:\mingw64\include\c++\12.2.0\bits\stl_pair.h:65,
16                   from c:\mingw64\include\c++\12.2.0\bits\stl_algobase.h:64,
17                   from c:\mingw64\include\c++\12.2.0\vector:60,
18                   from main.cpp:1:
19  c:\mingw64\include\c++\12.2.0\concepts: In substitution of 'template<class T>  re-
20  quires (random_access_range<T>) && (sortable<decltype(std::ranges::__cust_ac-
21  cess::__begin((declval<_Container&>)())), std::ranges::less, std::identity>) void
22  bubbleSort(T&) [with T = std::vector<Uncomparable>]':
23  main.cpp:31:15:   required from here
24  c:\mingw64\include\c++\12.2.0\concepts:336:13:   required for the satisfaction of
25  'invocable<_Fn, _Args ...>' [with _Fn = std::ranges::less&; _Args = {Uncompara-
26  ble&, Uncomparable&}]
27  c:\mingw64\include\c++\12.2.0\concepts:340:13:   required for the satisfaction of
28  'regular_invocable<_Fn, _Args ...>' [with _Fn = std::ranges::less&; _Args = {Un-
29  comparable&, Uncomparable&}]
30  c:\mingw64\include\c++\12.2.0\concepts:344:13:   required for the satisfaction of
31  'predicate<_Rel, _Tp, _Tp>' [with _Rel = std::ranges::less&; _Tp = Uncomparable&]
32  c:\mingw64\include\c++\12.2.0\concepts:349:13:   required for the satisfaction of
33  'relation<_Rel, _Tp, _Up>' [with _Rel = std::ranges::less&; _Tp = Uncomparable&;
34  _Up = Uncomparable&]
35  c:\mingw64\include\c++\12.2.0\concepts:359:13:   required for the satisfaction of
36  'strict_weak_order<_Fn&, typename std::__detail::__iter_traits_impl<typename
37  std::remove_cvref<_Tp2>::type, std::indirectly_readable_traits<typename std::re-
38  move_cvref<_Tp2>::type> >::type::value_type&, typename std::__de-
39  tail::__iter_traits_impl<typename std::remove_cvref<_I2>::type,
40  std::indirectly_readable_traits<typename std::remove_cvref<_I2>::type>
41  >::type::value_type&>' [with _Fn = std::ranges::less; _I1 = std::pro-
42  jected<__gnu_cxx::__normal_iterator<Uncomparable*, std::vector<Uncomparable,
43  std::allocator<Uncomparable> > >, std::identity>; _I2 = std::pro-
44  jected<__gnu_cxx::__normal_iterator<Uncomparable*, std::vector<Uncomparable,
45  std::allocator<Uncomparable> > >, std::identity>]
46  c:\mingw64\include\c++\12.2.0\bits\iterator_concepts.h:739:13:   required for the
47  satisfaction of 'indirect_strict_weak_order<_Rel, std::projected<_Iter, _Proj>,
48  std::projected<_Iter, _Proj> >' [with _Rel = std::ranges::less; _Iter =
49  __gnu_cxx::__normal_iterator<Uncomparable*, std::vector<Uncomparable, std::alloca-
50  tor<Uncomparable> > >; _Proj = std::identity]
51  c:\mingw64\include\c++\12.2.0\bits\iterator_concepts.h:914:13:   required for the
52  satisfaction of 'sortable<decltype (std::ranges::__cust_ac-
53  cess::__begin(declval<_Container&>())), std::ranges::less, std::identity>' [with T
54  = std::vector<Uncomparable, std::allocator<Uncomparable> >]
55  c:\mingw64\include\c++\12.2.0\concepts:336:25: note: the expression 'is_invoca-
56  ble_v<_Fn, _Args ...> [with _Fn = std::ranges::less&; _Args = {Uncomparable&, Un-
57  comparable&}]' evaluated to 'false'
58    336 |     concept invocable = is_invocable_v<_Fn, _Args...>;
59        |                         ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
60  ninja: build stopped: subcommand failed.
61
62
63
64
65
66
```

Najistotniejszym fragmentem błędu jest ostatni jego fragment: `note: the expression` **`'is_invocable_v<_Fn, _Args ...> [with _Fn = std::ranges::less&; _Args = {Uncomparable&, Uncomparable&}]'`** `evaluated to` **`'false'`** `concept invocable = is_invocable_v<_Fn, _Args...>;`. Można w nim przeczytać, że problemem jest niespełnienie Konceptu `invocable`, który wymaga, aby dana funkcja `_Fn` mogła być wywołana z argumentami `_Args...`. Typ `_Fn` to w tym przypadku `std::ranges::less` (czyli delegat do operatora <), a `_Args...` to w tym przypadku dwa argumenty typu `Uncomparable`.

Czy jest on czytelniejszy lub łatwiejszy do zinterpretowania niż błąd zaprezentowany w Listing 20, Listing 21 lub Listing 22?

Ciężko jest odpowiedzieć twierdząco na to pytanie. Jednym z czołowych atutów Konceptów miała być poprawiona jakość generowanych błędów. Faktycznie są one bardziej ustrukturalizowane i korzystają z nazwanych wymagań, zamiast raportować wyłącznie niekompilujące się linijki (oraz listować (czasami nawet setki) kandydatów, którzy **też nie pasują do wywołania**). Niestety praktyka pokazała, że nie zawsze skutkuje to czytelniejszymi i prostszymi do zinterpretowania błędami.

Warto przyjrzeć się przykładowi, który prezentuje CppReference w artykule poświęconym Konceptom i ograniczeniom (ang. *Constrains and concepts*) [33]. Traktuje on o błędach raportowanych przez kompilator przy próbie wywołania algorytmu `std::sort()` dla obiektu typu `std::list`. To wywołanie się nie powiedzie, ponieważ iteratory tej klasy nie są *random access*. Z założenia błąd przy `std::sort()` (które nie korzysta z Konceptów) powinien być zawiły, skomplikowany i niezrozumiały; mówiący głównie o nielegalnych operacjach wykonywanych na przekazanych iteratorach, takich jak odejmowanie czy dodawanie (co powinno się udać dla iteratorów *random access*). Natomiast błąd raportowany podczas użycia `std::ranges::sort()` (który korzysta z Konceptów) powinien być prosty i klarowny; mówiący jasno o tym, że nie można sortować zakresu na bazie iteratorów listy, ponieważ nie spełniają Konceptu `random_access_iterator`.

Niestety tak nie jest. Dokładny tekst tych błędów został pominięty, bo byłby bardzo podobny do tego z Listing 36. O ile Koncepty faktycznie pozwalają na dużo łatwiejsze tworzenia zestawów narzędzi, które mają operować na konkretnych grupach typów generycznych, to niestety zawiodły w dostarczeniu do języka C++ dużo wyższej jakości błędów kompilacji. Być może jest to kwestia niekompetencji autorów kompilatorów, lecz patrząc na poziom skomplikowania tego narzędzia, raczej jest to wina złych założeń dotyczących prostoty wykorzystania typów generycznych w C++.

4 Wpływ nowych elementów języka C++ dotyczących programowania generycznego na rozwój innych narzędzi

O ile Koncepty nie sprostały oczekiwaniom dotyczącym kreowania ich jako rozwiązania problematyczności związanej z poziomem skomplikowania treści błędów kompilacji w języku C++, to nie można im odmówić ogromnego wpływu na rozwój innego kluczowego narzędzia wprowadzonego w standardzie C++20 – Zakresów.

Termin „zakres" pojawił się już wcześniej w pracy, ale jego intuicyjna definicja dotycząca jakiegoś zbioru elementów lub jakiejś obszerności danych była wystarczająca do poprawnego zrozumienia kontekstu. Teraz warto jednak dokładniej zdefiniować to pojęcie z uwagi na wagę, którą narzędzie Ranges (z angielskiego – *Zakresy*) reprezentuje w Kompletnym C++. Gdy termin „zakres" jest pisany małą literą, reprezentuje on jakąś abstrakcję zbioru elementów. Gdy jednak ten termin jest pisany wielką literą, jednoznacznie referuje on do biblioteki Zakresów omawianej w tym rozdziale.

4.1 Zakresy

Biblioteka Zakresów [21] wprowadziła do języka ogromną liczbę narzędzi, dzięki którym znacznie prościej jest operować na zbiorach elementów, takich jak kontenery czy sekwencje generowane ad hoc.

Ich omówienie warto zacząć od najprostszego, choć jednocześnie najczęściej używanego nowowprowadzonego narzędzia. Tak zwanych *uzakresowionych algorytmów* (ang. *rangified algorithms*). Algorytmy te są stricte ulepszoną wersją algorytmów standardowych istniejących w bibliotece języka C++ od wielu lat – pozwalają one na wygodniejsze i bardziej elastyczne wykorzystywanie popularnych operacji na danych.

Jedną z najpopularniejszych operacji, jakie można wykonać na zbiorze danych, jest jego posortowanie. Do standardu C++17 włącznie, aby posortować daną kolekcję w standardowy sposób, należało skorzystać z szablonu `std::sort()`, na przykład tak, jak zostało to zaprezentowane w Listing 37.

Listing 37: Przykład użycia szablonu `std::sort()`.

```
1  auto main() -> int {
2      auto vector = std::vector<int>{3, 2, 1};
3      std::sort(vector.begin(), vector.end());
4  }
```

Większość programistów zaznajomionych z innymi językami programowania, widząc taki przykład, natychmiast zada dosyć oczywiste pytania:

- Co oznaczają `.begin()` oraz `.end()`?
- Czy naprawdę ich użycie jest tutaj konieczne?

Metody `.begin()` oraz `.end()`, które muszą być zdefiniowane dla wszystkich kolekcji, zwracają iteratory. Abstrakcja iteratora jest wykorzystywana w wielu językach programowania, w tym również w Javie oraz Pythonie. Pozwala ona na standardowe modelowanie zakresu danych, np.:

- całej kolekcji;
- fragmentu kolekcji;
- generowanego ad hoc zbioru danych;

> "Iterator jest konceptem używanym do wyrażania bieżącego miejsca w ciągu. Tak naprawdę początkowo iteratory miały być nazywane *współrzędnymi* lub *pozycjami* – można je uważać za uogólnienie wskaźników. W niektórych językach programowania iteratorami nazywa się masywne koncentracje funkcjonalności, lecz *koncepcja* iteratora wyraża jedynie to proste pojęcie pozycji."

~ Alexander Stepanov, [30], rozdział 10.4.

W każdym z przytoczonych wcześniej języków istnieją wbudowane narzędzia, które opierają się na tej abstrakcji. Przykładem może być pętla zakresowa `for()`, przedstawiona w Listing 38.

Listing 38: Przykład użycia pętli zakresowej (ang. *range-based* `for()` *loop*) w C++.

```
1  for (auto const& element : someCollection) { }
```

Aby obiekt typu `someCollection` mógł zostać użyty w ten sposób, musi on wspierać wywołanie metod `.begin()` oraz `.end()`[16], które zwrócą typ modelujący iterator. Typ modeluje iterator jeżeli, między innymi, wspiera operator preinkrementacji oraz dereferencji (`*`) (który zwróci jakąś wartość lub referencję).

Listing 39: Przykład użycia pętli zakresowej (ang. *for-each loop / enhanced for statement*) w Javie.

```
1  for (var element : someCollection) { }
```

W przypadku Javy typ obiektu `someCollection` musi implementować `interface Iterable`, aby można było go użyć w takiej pętli. Interfejs ten wymaga dostarczenia definicji metody `iterator()`, która musi zwracać obiekt typu implementującego `interface Iterator`[17].

[16] Jest to tak naprawdę niepoprawne uproszczenie. Formalnie typ nie musi wspierać tych metod. Jest to jednak uproszczenie, które aplikuje się do zdecydowanej większości przypadków. Pełna specyfikacja mówi o wyrażeniu *begin-expr* oraz *end-expr*, które, zależnie od typu `someCollection`, mogą albo delegować do arytmetyki wskaźników przy C-tablicach, albo do metod `.begin()` i `.end()`, albo szukać poprawnych wyrażeń `begin(someCollection)` oraz `end(someCollection)` za pomocą ADL (ang. *argument-dependent lookup*), czyli *przeszukiwania uzależnionego od argumentu*. [42]

[17] Nie należy mylić interfejsu `Iterator` z interfejsem `Iterable`.

Listing 40: Przykład użycia pętli zakresowej (ang. *for loop*) w Pythonie.

```
1  for element in someCollection:
```

W bardzo podobny sposób są sprecyzowane wymagania w przypadku języka Python. Typ, który chce wspierać używanie go w kontekście pętli, musi implementować metodę `.__iter__()`, która musi zwracać typ modelujący iterator. Oznacza to, że ten z kolei typ ten musi implementować metodę `.__next__()` oraz (ponownie) `.__iter__()`. W tym przypadku ta druga metoda zwykle zwraca obiekt, na którym jest wywoływana (`self`).

Jak można zauważyć, każdy z tych języków silnie integruje wbudowane narzędzie z abstrakcją iteratora. Dodatkowo, niektóre bardziej skomplikowane narzędzia również korzystają z abstrakcji iteratorów. Przykładami są strumienie z Javy 8 (ang. *Java 8 Streams*) czy adaptery zakresowe (ang. *range-adaptors*) z C++20, które będą omówione w dalszej części rozdziału.

Wracając do omawiania Listing 37, należy teraz odpowiedzieć na drugie zadane pytanie. Czy użycie tych metod jest konieczne? Czy nie można „zwyczajnie" przekazać całego obiektu `vector` do algorytmu? Jaki jest powód, dla którego algorytm sortowania pracuje wyłącznie na iteratorach, a nie na całych kolekcjach?

Odpowiedź leży w – oryginalnie – dosyć przekonywującym argumencie. Patrząc z perspektywy abstrakcji, przekazanie dowolnych pozycji (iteratorów) jest bardziej generycznym / uogólnionym przypadkiem pracy, niż przekazanie całej kolekcji. W końcu cała kolekcja to też jakiś podzakres (tak samo jak w matematyce dany zbiór jest swoim własnym podzbiorem).

Niestety, czasem realny świat weryfikuje, że piękne i eleganckie tezy matematyczne negatywnie wpływają na praktyczne doświadczenia. Efektywnie, programiści najczęściej chcą zaaplikować daną operację do całej kolekcji, a nie do jej fragmentu. Z jednej strony lepiej jest mieć możliwość sprecyzowania zarówno całego, jak i fragmentu danego zakresu, niż tej możliwości nie mieć. Z drugiej strony języki uzbrojone w narzędzia takie jak przeciążanie funkcji (albo szablonów funkcji) mogą wykorzystać te przewagę i zdefiniować dwa interfejsy dla funkcji – jeden w postaci przeciążenia przyjmującego dwa iteratory i drugi w postaci przeciążenia przyjmującego całą kolekcję[18].

Kontrargumentem ku sprecyzowaniu dwóch (lub więcej) takich interfejsów jest fakt istnienia iteratorów, które w sposób leniwy zmieniają porządek kolekcji. Przykładem są iteratory odwracające (ang. *reverse iterators*), dzięki którym można przejść przez elementy kolekcji w odwrotnej kolejności. Takich iteratorów może być więcej. Oznacza to, że trzeba by zdefiniować więcej interfejsów pracujących na różnych „widokach" zakresów. Dobrym przykładem użycia takiego iteratora jest optymalna implementacja funkcji, która weryfikuje, czy przekazany zakres (np. `std::string`) jest palindromem, czyli czytany „od lewej do prawej" i „od prawej do lewej" wygląda tak samo.

[18] Alternatywnie można też założyć, że każda kolekcja powinna być w stanie zwracać jakiś „widok" na swój fragment. Przykładem może być metoda `.sublist()` z interejsu `List<T>` z Javy.

Listing 41: Przykład implementacji funkcji weryfikującej, czy przekazany tekst jest palindromem.

```
1 auto isPalindrome(std::string const& str) -> bool {
2     return std::equal(
3             str.begin(), str.begin() + str.size() / 2,
4             str.rbegin()
5     );
6 }
```

Idea implementacji zaprezentowanej w Listing 41 jest bardzo prosta. Algorytm sprawdza, czy dwa zakresy są sobie równe. Pierwszy z nich (zdefiniowany przez parę iteratorów `str.begin(), str.begin() + str.size() / 2`) reprezentuje pierwszą połową tekstu `str`. Drugi z nich rozpoczyna się od końca `str`, ale jego kolejne elementy to tak naprawdę przedostatni, przed-przedostatni itd. Logika patrzenia na zakres w odwróconej kolejności jest zawarta w iteratorze odwracającym, który został zwrócony przez metodę `.rbegin()`[19].

Przeszkoda w postaci niewygodnej pracy na pełnych zakresach okazała się być jednak do pokonania. Wraz z C++20 wprowadzono bibliotekę Zakresów, której pierwsze narzędzie omówione w tej pracy to algorytmy zakresowe.

4.1.1 Algorytmy zakresowe

Listing 37 przedstawia relatywnie niewygodne użycie algorytmu sortowania. Dzięki Zakresom z C++20 można ten kod uprościć w sposób zaprezentowany w Listing 42.

Listing 42: Przykład użycia `std::ranges::sort()`.

```
1 auto main() -> int {
2     auto vector = std::vector<int>{3, 2, 1};
3     std::ranges::sort(vector);
4 }
```

Zakresy wprowadziły do języka nową przestrzeń nazw – `ranges`. Zawiera ona ulepszone „kopie" dotychczas dostępnych algorytmów standardowych[20]. Nowe wersje algorytmów są stricte lepsze. Nie istnieją powody, dla których programista chciałby korzystać ze starszych wersji. Nawet jeżeli potrzebuje przekazać jakiś fragment zakresu (a nie całą kolekcję), to wersja z `ranges` wciąż będzie lepsza. Listing 43 prezentuje przykłady pracy na pierwszych trzech elementach danego wektora.

[19] Należy zwrócić uwagę na różnicę między `.begin()` oraz `.rbegin()`. Dotyczy ona pierwszej litery metody `.rbegin()` to *reverse-begin*.

[20] W momencie pisania pracy, nie wszystkie algorytmy standardowe doczekały się swojej wersji w `ranges`. Przede wszystkim algorytmy z nagłówka `<numeric>` nie mają swoich uzakresowionych odpowiedników.

Listing 43: Przykład użycia standardowego algorytmu sortowania wyłącznie na pierwszych trzech elementach wektora.

```
1  auto main() -> int {
2      auto numbers = std::vector<int>{5, 4, 3, 2, 1};
3
4      std::sort(numbers.begin(), numbers.begin() + 3);          // 1
5
6      auto firstThree = std::ranges::subrange(
7              numbers.begin(), numbers.begin() + 3
8      );
9      std::ranges::sort(firstThree);                            // 2
10
11     std::ranges::sort(numbers.begin(), numbers.begin() + 3);  // 3
12 }
```

Jak widać, efektywnie istnieją trzy standardowe sposoby na osiągnięcie sprecyzowanego celu:

1. Wywołanie standardowego, „starego" `std::sort()`. Wymagane jest przekazanie pary iteratorów.

2. Wywołanie „nowego" `std::ranges::sort()`. Pierwsze trzy elementy są reprezentowane przez podzakres (obiekt typu `std::ranges::range<IteratorType>`) realizowany jako opakowanie na parę iteratorów. W przypadku, gdy chce się później wykonać więcej operacji na tym podzakresie, dobrym pomysłem może być zapisanie go do zmiennej, jak w prezentowanym przypadku.

3. Jeżeli jednak takiej potrzeby nie ma, to warto skorzystać z faktu, że `std::ranges::sort()` wspiera pracę z przekazanymi iteratorami – tak samo jak klasyczny `std::sort()`.

Warto jeszcze poruszyć temat potencjalnej wydajności wersji zakresowej. Algorytm przyjmujący cały zakres (tak jak w Listing 42) posiada więcej informacji o samym typie, niż algorytm, który otrzymał wyłącznie iteratory do niego (tak jak w Listing 37). Oznacza to, że taki algorytm może czasami wydajniej wykonać zleconą pracę. Niektóre przypadki są jeszcze bardziej skrajne – algorytm, który nie ma dostępu do całego zakresu, może w ogóle nie być w stanie wykonać swojej pracy.

Przykładem jest sortowanie obiektu typu `std::list`. Jest to lista pojedynczo wiązana (ang. *singly-linked-list*). Oznacza to, że jej iteratory nie są *random access* (czyli nie wspierają indeksowania dowolnego elementu w stałym czasie). Nie implikuje to jednak tego, że takiej listy nie można posortować. Istnieją algorytmy sortowania takich list (np. algorytm sortowania przez scalanie – ang. *merge sort*), a `std::list`, będąc świadomym swoich szczegółów implementacyjnych, może taką operację wspierać przy pomocy metody – `.sort()`.

Oczywiście z uwagi na swoją (przytoczoną w Listing 33) sygnaturę, `std::ranges::sort()` nie pozwoli na posortowanie takiej listy, ale gdyby można było to narzędzie rozszerzyć o przeciążenie zaprezentowane w Listing 44, to problem by zniknął.

Listing 44: Hipotetyczne przeciążenie narzędzia `sort()`[21].

```
1  template <std::ranges::range Range>
2  requires requires(Range& range) { range.sort(); }
3  auto sort(Range& range) -> decltype(auto) {
4      return range.sort();
5  }
```

Przedstawiony kod pozwoliłby na ujednolicenie interfejsu sortowania. Gdyby można było rozszerzyć przestrzeń nazw `ranges` o taką funkcjonalność, to `std::ranges::sort()` mógłby działać nie tylko dla zakresów, które oferują *random access* do swoich elementów, ale też i dla **wszystkich** typów, które potrafią „posortować siebie".

Warto zwrócić uwagę, jak proste jest sprecyzowanie i dodanie takiego narzędzia. Wymaga ono dostarczenia jakiegoś zakresu, na którego nałożono dodatkowe wymaganie – musi on udostępniać metodę `.sort()`. Jedyne co w takiej sytuacji robi algorytm, to deleguje całą pracę do wywołania wcześniej wspomnianej metody.

Takie rozszerzenie byłoby niesamowicie trudne do stworzenia bez używania elementów wprowadzonych w C++20. Wymagałoby integracji z istniejącymi narzędziami realizowanej za pomocą skomplikowanego metaprogramowania wykorzystującego mechanikę SFINAE, o której była mowa wcześniej w pracy.

4.1.2 Projekcje

Projekcje są wygodnym sposobem na adaptowanie predykatu, komparatora czy innej funkcji danego algorytmu.

Listing 45 przedstawia wywołania algorytmów sortujących, które cechują się podobieństwem na zasadzie brania pod uwagę jakiejś **pojedynczej** cechy sortowanego typu.

Listing 45: Sortowanie różnych typów z uwagi na ich pojedynczą cechę.

```
1  #include <algorithm>
2  #include <cmath>    // std::hypot
3  #include <string>
4  #include <vector>
5
6  struct Protagonist {
7      std::string name;
8  };
9
10 struct Point {
11     int x;
12     int y;
```

[21] Choć może się to wydawać nieco dziwne, to składnia `requires requires` jest tu celowa i poprawna. Pierwsze `requires` definiuje klauzulę, czyli ograniczenie dla danego typu (lub typów) szablonowego. Drugie `requires` precyzuje wyrażenie, które określa wymagane operacje.

```
13  };
14
15  auto main() -> int {
16      using std::vector;
17      using std::string;
18      using std::ranges::sort;
19
20      auto protagonists = vector<Protagonist>{
21          {"Jonathan"}, {"Joseph"}, {"Jotaro"}, {"Josuke"}, {"Giorno"}
22      };
23
24      auto points = vector<Point>{{1, 2}, {0, 5}, {0, 3}, {4, 0}};
25
26      auto languages = vector<string>{
27          "Python", "Java", "C++", "Kotlin", "C#", "Objective-C"
28      };
29
30      sort(
31              protagonists,
32              [](Protagonist const& left, Protagonist const& right) {
33                  return left.name < right.name;
34              }
35      );
36
37      sort(points, [](Point const left, Point const right) {
38          auto const leftDistance  = std::hypot(left.x, left.y);
39          auto const rightDistance = std::hypot(right.x, right.y);
40          return leftDistance < rightDistance;
41      });
42
43      sort(languages, [](string const& left, string const& right) {
44          return left.size() < right.size();
45      });
46  }
```

W tym przykładzie, protagoniści są sortowani alfabetycznie po swoich imionach. Punkty są sortowane rosnąco w zależności od ich odległości od punktu $(0,0)$, a nazwy języków programowania są sortowane od najkrótszej do najdłuższej.

Warto zwrócić uwagę na fakt, że sprecyzowane w formie lambd komparatory nie robią nic innego, jak porównanie „wyciągniętych" cech z danych typów. Efektywnie:

- Porównywanie protagonistów *sprowadza się* do porównywania ich ***imion***.
- Porównywanie punktów *sprowadza się* do porównywania ich **odległości od punktu $(0,0)$**.
- Porównywanie języków (obiektów typu `std::string`) *sprowadza się* do porównywania ich **długości**.

Jeżeli porównywanie danego typu *sprowadza się* do porównywania jakiejś jego **cechy**, to wykorzystanie komparatora można zastąpić wykorzystaniem projekcji. Projekcja pozwoli operować (zwykle domyślnemu) komparatorowi na interesującej programisty **cesze**, co spowoduje, że dane typy zostaną posortowane z uwagi na ww. cechę. Operacje sortujące z Listing 45 można zastąpić bardziej zwięzłymi alternatywami, zaprezentowanymi w Listing 46.

Listing 46: Przykład zastąpienia komparatorów projekcjami.

```
sort(protagonists, {}, [](Protagonist const& p) {
    return p.name;
});

sort(points, {}, [](Point const p) {
    return std::hypot(p.x, p.y);
});

sort(languages, {}, [](string const& s) {
    return s.size();
});
```

Zamiast używać komparatorów, zostawiane są ich domyślne wartości (literał `{}` – dzięki kodzie zaprezentowanym w Listing 33 wiadomo, że jest to obiekt typu `std::ranges::less`). Precyzowane są natomiast projekcje (trzeci argument `sort()`). Algorytm będzie sortować elementy przekazanego zakresu według komparatora, który jednak nie będzie otrzymywał (i porównywał) obiektów źródłowych, a jedynie wyniki projekcji zaaplikowanej na tych obiektach.

Dla kompletności i dokładnego zrozumienia interakcji komparatorów i projekcji, warto przyjrzeć się generycznemu algorytmowi `bubbleSort()` (Listing 35), który został wzbogacony o możliwość sprecyzowania tych dwóch narzędzi (Listing 47).

Listing 47: Generyczny algorytm `bubbleSort()` wspierający precyzowanie komparatora i projekcji.

```
template<
        std::ranges::random_access_range Range,
        typename Comparator = std::ranges::less,
        typename Projection = std::identity>
requires std::sortable<
        std::ranges::iterator_t<Range>,
        Comparator,
        Projection
>
auto bubbleSort(
        Range& container,
        Comparator comparator = {},
        Projection projection = {}
) -> void {
    auto n = container.size();

    for (auto i = 0; i < n - 1; i++) {
        for (auto j = 0; j < n - i - 1; j++) {
            auto const shouldSwap = comparator(
                    projection(container[j + 1]),
                    projection(container[j])
            );
            if (shouldSwap) {
                std::swap(container[j + 1], container[j]);
            }
        }
```

```
27      }
28  }
```

Może się to wydawać nieprawdopodobne, ale algorytmy z `ranges` pozwalają uprościć kod z Listing 46 jeszcze bardziej. To dlatego, że do wywołania projekcji i komparatora nie używają prostej notacji wywołania funkcji, czyli `nazwaFunkcji(argumenty)`, a pomocniczego szablonu `std::invoke()` [34]. Co to zmienia? Chociażby to, że wspierana jest notacja wskaźnika na pole (ang. *pointer to member*) w sposób zaprezentowany w Listing 48.

Listing 48: Zastąpienie niektórych projekcji wskaźnikami do pól[22].

```
1  sort(protagonists, {}, &Protagonist::name);
2
3  sort(points, {}, [](Point const p) {
4      return std::hypot(p.x, p.y);
5  });
6
7  sort(languages, {}, &std::string::size);
```

W tym momencie warto przyjrzeć się, jak projekcje wyglądają w innych, omawianych w pracy językach. Język Java nie wspiera abstrakcji projekcji jako swojego rodzaju mapowania argumentów przy porównywaniu ich podczas wykonywania algorytmu sortowania. Czy to oznacza, że „klasyczne" użycie komparatorów zaprezentowane w Listing 49 to jedyny sposób na osiągnięcie celu?

Listing 49: Tożsama logika sortowania kolekcji z Listing 45 w języku Java.

```
1   Collections.sort(
2           protagonists,
3           (left, right) -> left.name.compareTo(right.name)
4   );
5
6   Collections.sort(
7           points,
8           (left, right) -> {
9               var leftDistance = Math.hypot(left.x, left.y);
10              var rightDistance = Math.hypot(right.x, right.y);
11              return Double.compare(leftDistance, rightDistance);
12          }
13  );
14
15  Collections.sort(
16          langauges,
17          (left, right) -> Integer.compare(
18                  left.length(), right.length()
19          )
20  );
```

[22] Formalnie składnia `&std::string::size` skutkuje wywołaniem zachowania niesprecyzowanego (nie mylić z niezdefiniowanym (UB)). Przykład ten jednak został zachowany ze względu na prezentację możliwości sprecyzowania wskaźnika do metody, a nie tylko do pola.

Odpowiedź na zadane przed chwilą pytanie brzmi – nie. Podany w Listing 49 sposób nie jest jedynym, za pomocą którego można osiągnąć cel. Zamierzony efekt można osiągnąć korzystając ze składni, która jest bardzo podobna do projekcji z języka C++. Została ona zaprezentowana w Listing 50.

Listing 50: Logika sortowania z Listing 49 wykorzystująca metody-fabryki komparatorów.

```
1  Collections.sort(
2          protagonists,
3          Comparator.comparing(Protagonist::getName)
4  );
5
6  Collections.sort(
7          points,
8          Comparator.comparingDouble(point -> Math.hypot(point.x, point.y))
9  );
10
11 Collections.sort(
12         langauges,
13         Comparator.comparingInt(String::length)
14 );
```

Listing 50 prezentuje kod, który miejscami niewątpliwie przypomina zaprezentowane projekcje z Listing 48. Nie jest to jednak to samo narzędzie. W języku C++ projekcje i komparatory są osobnymi bytami, a w języku Java zostały one złączone w pojedyncze narzędzie. Język Java dostarcza metody-fabryki tworzące komparatory na podstawie logiki sprecyzowanej projekcji.

W języku Python sytuacja jest zależna od jego wersji. Obecnie stosowaną praktyką jest wykorzystanie tak zwanej funkcji klucza (ang. *key function*). Jest ona zwyczajną projekcją – w rozumieniu bliskim temu z języka C++.

Listing 51: Tożsama logika sortowania kolekcji z Listing 45 w języku Python.

```
1  protagonists.sort(key=lambda protagonist: protagonist.name)
2
3  points.sort(key=lambda point: math.hypot(point.x, point.y))
4
5  languages.sort(key=lambda language: len(language))
```

Listing 51 ukazuje oczywiste podobieństwo między domyślnym sposobem sterowania logiką sortującą a projekcjami z języka C++. Co więcej, wiele źródeł (między innymi [35] i [36]) podaje, że obecne praktyki Pythonowe (od wersji 3.0) stanową kompletną odwrotność tych z Javy. W Javie projekcje były używane do stworzenia komparatorów. Nie istniały jako osobne, samowystarczalne w kontekście algorytmów sortowania byty. Ale w Pythonie, chcąc użyć komparatora, należy stworzyć z niego projekcję i dopiero ona będzie użyta do sterowania zachowaniem algorytmu.

Wraz z wprowadzeniem projekcji do C++20 udało się dodatkowo naprawić pewne niespójności wśród silnie skojarzonych ze sobą algorytmów, którym dedykowany jest kolejny przykład.

Rozważmy typ opakowujący pojedynczą zmienną, np. typu `int`. Następnie, stworzona kolekcja takich obiektów zostanie posortowana zgodnie z naturalnym porządkiem wspomnianego pola. Później zostanie użyty algorytm wyszukiwania binarnego, który znajdzie pojemnik opakowujący pole o danej wartości.

Rozważany typ nie będzie porównywalny sam w sobie – nie będzie implementował operatora `<` (ani wprowadzonego w C++ `<=>`). Konieczne jest zatem sprecyzowanie komparatora (lub projekcji), dzięki któremu będzie można te obiekty posortować oraz wyszukiwać binarnie. Nie jest to jednak trywialne, jak można zauważyć analizując Listing 52 i Listing 53.

Listing 52: Sortowanie kolekcji domyślnie nieporównywalnego typu przy pomocy `std::sort()` oraz komparatora[23].

```
 1  struct Wrapper {
 2      int field;
 3  };
 4
 5  auto main() -> int {
 6      auto wrappers = std::vector<Wrapper>{
 7              Wrapper(3), Wrapper(1), Wrapper(4),
 8              Wrapper(2), Wrapper(5), Wrapper(0)
 9      };
10
11      auto const fieldComparator =
12              [](Wrapper const left, Wrapper const right) {
13                  return left.field < right.field;
14              };
15
16      std::sort(
17              wrappers.begin(), wrappers.end(),
18              fieldComparator
19      );
20  }
```

Teraz warto przyjrzeć się, jak by wyglądała konstrukcja pozwalająca użyć wyszukiwania binarnego mająca na celu sprawdzenie, czy we `wrappers` znajduje się obiekt o wartości pola `field` równemu `2`.

Listing 53: Wyszukiwanie binarne w posortowanej kolekcji typów nieporównywalnych domyślnie przy pomocy klasycznego komparatora.

```
1  auto const wrappedTwo = std::lower_bound(
2          wrappers.begin(), wrappers.end(),
3          Wrapper(2),
4          fieldComparator
5  );
```

Kod z Listing 53 jest prosty i przejrzysty – używa tego samego komparatora, który został wykorzystany do posortowania kolekcji. Niestety nie jest to zawsze optymalne rozwiązanie. Warto zwrócić uwagę, że

[23] Warto zwrócić uwagę na użycie nawiasów okrągłych przy konstrukcji obiektów typu `Wrapper`. Ich użycie w kontekście inicjalizacji typów agregacyjnych jest jednym z nowości wprowadzonych w C++20 [48].

w linijce nr 3 tworzony jest tymczasowy obiekt, którego jedynym celem jest bycie porównywanym z elementami z `wrappers`. Po tym procesie obiekt ten będzie zniszczony. W przypadku lekkich typów (ang. *lightweight types*) (czyli takich, które zajmują mało pamięci) nie stanowi to problemu. Co jednak, gdyby `Wrapper` był kosztownym do stworzenia typem? Przede wszystkim `fieldComparator` by przyjmował te obiekty przez stałą referencję, a nie przez stałą kopię[24], ale nie zmieniłoby to faktu, że aby kod z omawianego listingu zadziałał, trzeba by stworzyć tymczasowy obiekt.

Alternatywnym rozwiązaniem do tego, które zostało zaprezentowane w Listing 53 jest skorzystanie z tak zwanego heterogenicznego komparatora. Zamiast tworzenia tymczasowego obiektu typu `Wrapper`, algorytm będzie porównywał elementy kolekcji z liczbą 2 w sposób zaprezentowany w Listing 54.

Listing 54: Wyszukiwanie binarne w posortowanej kolekcji typów nieporównywalnych domyślnie przy pomocy heterogenicznego komparatora.

```
1  auto const wrappedTwo = std::lower_bound(
2          wrappers.begin(), wrappers.end(),
3          2,
4          [](Wrapper const w, int const n) {
5              return w.field < n;
6          }
7  );
```

Tak usprawniony kod jest optymalny – nie wymaga niepotrzebnego tworzenia tymczasowego obiektu. Cechuje go jednak inny problem – sprecyzowany komparator jest niespójny z tym, który został zaprezentowany w Listing 52 w celu posortowania kolekcji (`fieldComparator`). Jest to przykład poważnej niekonsekwentności. Komparator ustalający porządek według danego pola (`.field`) również powinien być w stanie zostać użyty przy wyszukiwaniu binarnym bazującym na identycznym porządku elementów.

Autor niniejszej pracy przyglądał się temu problemowi podczas procesu przenoszenia kodu z C++17 na C++20, między innymi zamieniając starsze algorytmy na wersje z `ranges`. Zaobserwował, że zwyczajne zamienienie nazwy `std::lower_bound` na `std::ranges::lower_bound` poskutkowało błędem kompilacji. Zadał później pytanie na platformie StackOverflow, które doczekało się odpowiedzi wskazującej na zmianę interfejsu, która naprawiła te niespójność [37]. Okazuje się, że projekcje są w stanie zunifikować interfejs adaptujący odpowiednie zachowania [38] w sposób przedstawiony w Listing 55.

[24] W zaprezentowanym przykładzie rozważane komparatory przyjmują swoje argumenty przez kopię z uwagi na to, że przekazywanie małych obiektów przez referencję jest znaną pesymizacją. Nie na tym jednak należy się koncentrować – uwagę powinno się zwrócić ku problemach z istnieniem dwóch różnych rodzajów komparatorów, z których jeden jest niekompatybilny z algorytmem sortowania.

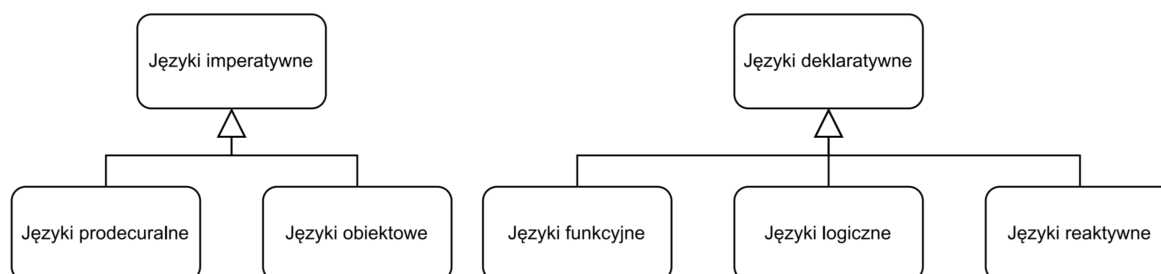
Listing 55: Sortowanie i wyszukiwanie binarne w kolekcji przy pomocy narzędzi zakresowych z C++20.

```
struct Wrapper {
    int field;
};

auto main() -> int {
    auto wrappers = std::vector<Wrapper>{
            Wrapper(3), Wrapper(1), Wrapper(4),
            Wrapper(2), Wrapper(5), Wrapper(0)
    };

    auto const byField = &Wrapper::field;

    std::ranges::sort(wrappers, {}, byField);

    auto const wrappedTwo = std::ranges::lower_bound(
            wrappers, 2, {}, byField
    );
}
```

Jak widać w Listing 55, dzięki narzędziom z `ranges` programista jest w stanie wykorzystać tą samą logikę zarówno do sprecyzowania porządku elementów w celu ich posortowania, jak i do sprecyzowania porządku elementów w celu wyszukania któregoś z nich przy pomocy wyszukiwania binarnego. Jest to krok w stronę większej spójności, konsekwentności i łatwości wykorzystywania narzędzi języka.

4.1.3 Adaptery zakresów

Istnieje wiele paradygmatów języków programowania, wśród których można wyróżnić następującą (niekompletną) hierarchię:



Rysunek 3: Przykładowa hierarchia paradygmatów języków programowania.

Większość współczesnych języków programowania cierpi na podobny problem klasyfikacji w kontekście paradygmatu jak i w kontekście poziomu abstrakcji, o którym była mowa w rozdziale **3.4**. Dobrym przykładem jest Java, która mimo tak silnie zintegrowanego paradygmatu obiektowego, nie może być nazwana ani w pełni obiektowym językiem (np. z uwagi na istnienie prymitywów), ani językiem, który nie oferuje konstrukcji z innych paradygmatów (np. z uwagi na istnienie typów generycznych, refleksji czy referencji do metod).

Hierarchia przedstawiona w Rysunek 3 jest z tego powodu bardzo mało użyteczna. Niemniej jednak warto mieć świadomość istnienia różnic i motywacjom ku adaptowaniu różnych paradygmatów.

Jako przykład analizy takiej adaptacji mogą posłużyć adaptery zakresów wprowadzone w C++20, które wprowadzają do świata języka C++ podobne ułatwienia do tych, które przyniosły ze sobą strumienie z Javy 8 [39].

4.1.3.1 Paradygmat funkcyjny

Paradygmat funkcyjny, z uwagi na bycie szczególnym przypadkiem paradygmatu deklaratywnego, cechuje się precyzowaniem *jakie* operacje mają być wykonane, zamiast koncentrować się na tym, *jak* one mają zostać zrealizowane.

Pozwala on programiście skoncentrować się na sprecyzowaniu wysokopoziomowych operacji osiągających dany cel, zamiast wymuszać przejmowanie się szczegółami implementacyjnymi. Można zauważyć tutaj podobieństwo do praktyki programowania do interfejsów – tam też należy odciągnąć uwagę od konkretnych implementacji i kierować ją ku temu, *co* dany obiekt może zrobić, a nie *jak*.

Jako przykład posłuży proste zadanie polegające na wyświetleniu co drugiej potęgi liczby 2, ograniczając się do pierwszych dziesięciu wygenerowanych wartości. Podejście proceduralne korzysta z narzędzi (relatywnie) niższego poziomu, które dokładnie określają listę kroków potrzebnych do osiągnięcia celu. Innymi słowy, paradygmat proceduralny (ale też i strukturalny, a zatem również i imperatywny) koncentruje się na tym, *jak* osiągnąć dany cel, zamiast precyzować *co* jest danym celem.

W ramach pierwszego przykładu, zaprezentowany został kod w języku Java (Listing 56 i Listing 57). Wybór ten jest celowy – narzędzia programowania funkcyjnego z Javy, z którymi będzie porównywane podejście proceduralne, są dostępne od dłuższego czasu, dzięki czemu zakłada się, że są lepiej znane od tych wprowadzonych w C++20. Porównanie wykorzystujące narzędzia Javowe pozwoli zatem na lepsze zrozumienie odpowiedników z C++.

Listing 56: Proceduralne rozwiązanie problemu generowania wybranych potęg dwójki w języku Java.

```
1  for (int i = 0, counter = 0; counter < 10; i++) {
2      if (i % 2 == 0) {
3          System.out.println((int)Math.pow(2, i));
4          ++counter;
5      }
6  }
```

Jako alternatywne rozwiązanie posłuży kod wykorzystujący strumienie z Javy 8, zaprezentowany w Listing 57.

Listing 57: Funkcyjne rozwiązanie problemu generowania wybranych potęg dwójki w języku Java.

```
1  IntStream.iterate(0, i -> i + 1)
2          .filter(i -> i % 2 == 0)
3          .map(i -> (int)Math.pow(2, i))
4          .limit(10)
5          .forEach(System.out::println);
```

Wbrew pozorom dla czytelnika niezaznajomionego z narzędziami programowania funkcyjnego (w dowolnym języku) wcale nie jest oczywiste, dlaczego kod z Listing 57 powinien być preferowany nad tym z Listing 56. Tematem niniejszej pracy nie jest dokładna analiza paradygmatów programistycznych, dlatego argumentacja popierająca tezę traktującą o przewadze podejścia funkcyjnego została ograniczona do minimum.

Aby zrozumieć kwintesencję omawianej przewagi, należy zidentyfikować kilka podstawowych lematów:

1. Należy rozróżniać *prostotę* / *łatwość* od *znajomości* (ang. *simple* / *easy versus familiar*). Coś, co jest nowe, jest niezrozumiałe, dlatego początkujący programista może pierwotnie traktować kod z Listing 56 jako lepszy z uwagi na to, że jest on (w mniemaniu tego programisty) prostszy. Nie jest to jednak prawda – taki tok rozumowania myli *nieskomplikowanie* z *niezaznajomieniem*. Aby móc obiektywnie ocenić wady i zalety oraz porównać obydwa podejścia, wpierw trzeba być dobrze zaznajomiony z obydwoma. Należy tutaj też pamiętać, że trudność opanowania danego narzędzia przekłada się jedynie pośrednio na całokształt jego poziomu skomplikowania. Narzędzie trudne do opanowania jest w pewien sposób trudniejsze od narzędzia łatwiejszego do opanowania, lecz w większości przypadków te różnice są zbyt małe, aby taka argumentacja miała wartość merytoryczną.

2. Pętla `for()`, o ile jest „bardziej podstawowa", jest – wbrew pozorom – trudniejsza do analizy. Jej nazwa w żadnym stopniu nie wskazuje na jej przeznaczenie. Konstrukcja ta wspiera arbitralne warunki wykonania, które mogą być przerwane przy pomocy takich słów kluczowych jak `break` czy `continue`. Należy dokładnie przeanalizować **każdy** jej element (wraz z całym ciałem), aby zidentyfikować, jaki jest jej cel. Innymi słowy, aby dowiedzieć się, *co* jest osiągane, trzeba wpierw zidentyfikować, *jak* dany cel jest osiągany.

3. Argument z punktu 2 można również zaaplikować do dowolnej, podstawowej instrukcji sterującej. `if()`, `while()` czy `switch()` nie mają nazw, które reprezentują ich przeznaczenie. Między innymi dzięki temu należy tworzyć dobrze nazwane, małe metody / funkcje, które pozwolą nadać nazwę tym konstrukcjom.

Warto teraz przyjrzeć się implementacji z Listing 57 mając na uwadze wymienione uwagi. Kod ten zastępuje nazwanymi instrukcjami nienazwane, prymitywne operacje sterujące. Te nazwane instrukcje nie precyzują sposobu ich realizowania. Analiza tego kodu przestaje wymagać objęcia umysłem całokształtu. Zamiast tego pozwala na interpretowanie operacji precyzujących cel jedna po drugiej. Czytając ten kod linijka po linijce można go bez problemu zidentyfikować:

1. iterując od zera kolejnymi liczbami naturalnymi;

2. biorąc pod uwagę wyłącznie liczby podzielne przez 2 (czyli co drugą liczbę);
3. traktuj każdą z nich jako wykładnik i wyprodukuj wartość 2 podniesioną do tej potęgi;
4. ogranicz się do pierwszych dziesięciu otrzymanych wartości;
5. a na koniec wyświetl wszystkie z nich.

Naturalnie nie każdy problem da się rozwiązać za pomocą strumieni w elegancki sposób. Mimo tego należy mieć świadomość przewagi kodu wykorzystującego nazwane operacje, które pozwalają w prosty sposób definiować cel, nad narzędziami, które służą jedynie opisaniu, jak dany cel chce się osiągnąć.

4.1.3.2 Widoki

Strumienie z Javy 8 odniosły ogromny sukces i zdobyły uznanie programistów, którym deklaratywne konstrukcje pozwoliły na tworzenie (z założenia) poprawnych rozwiązań (wspomniane na początku rozdziału **3.4.1**). C++20, wraz z uzakresowionymi algorytmami, wprowadził narzędzie bliźniacze do tego, które powitało świat w Javie 8. Nazywa się ono *adaptery zakresów*.

Adaptery zakresów są modyfikatorami, które aplikuje się do widoków (ang. *views*). *Widok* to abstrakcja nad zakresem, zdefiniowana za pomocą następujących Konceptów i narzędzi wspomagających.

Listing 58: Koncept `view`, który musi spełniać każdy widok.

```
1  template <typename T>
2  concept view = std::ranges::range<T>
3          and std::movable<T>
4          and std::ranges::enable_view<T>;
```

Jak widać w Listing 58, każdy widok musi być zakresem (szczegóły w przypisie 14), musi współpracować z semantykami przenoszenia (wprowadzonymi w C++11) oraz jego twórca musi jawnie zaznaczyć, że chce, aby tworzony typ nim był. Realizowane jest to za pomocą specjalizacji zmiennej szablonowej (ang. *variable template*) (dostępne od C++14 [40]) `enable_view`, przedstawionej w Listing 59. Taki sposób jawnego spełniania danego Konceptu może się kojarzyć z implementowaniem interfejsów w Javie – tam również twórca klasy musi jawnie zaznaczyć, który typ implementuje dany `interface`.

Listing 59: Zmienna szablonowa `enable_view`.

```
1  template <typename T>
2  inline constexpr bool enable_view =
3          std::derived_from<T, view_base> or /*is-derived-from-view-inter-
   face*/<T>;
```

gdzie `view_base` to pusta klasa służąca jedynie do reprezentowania danego interfejsu. Per CppReference [41], *is-derived-from-view-interface* jest spełnione jeżeli `T` “ma wyłącznie jedną, publiczną klasę bazową typu `std::ranges::view_interface<U>` dla dowolnego `U` […].”

Cały ten zestaw wprowadza programistę w świat widoków i ich adapterów. Przykładowym podstawowym adapterem jest `take`, zaprezentowany w Listing 60.

Listing 60: Przykład użycia adaptora `std::views::take` zwracającego widok na pierwsze trzy elementy wektora.

```
1  #include <iostream>
2  #include <ranges>   // views
3  #include <vector>
4
5  auto main() -> int {
6      auto numbers = std::vector<int>{1, 2, 3, 4, 5};
7
8      for (auto const n : numbers | std::views::take(3)) {
9          std::cout << n << ' ';
10     }
11 }
```

W przykładzie w Listing 60 najważniejszym fragmentem kodu jest operator *pipe* (ang. *pipe* – rura, przewód – `operator |`) i następujący po nim adapter `std::views::take`. Kod ten wyświetli: `1 2 3 `. Jest to bardzo prosty sposób na pracę z fragmentem jakiegoś zakresu.

Zanim przejdzie się do dokładnych wyjaśnień poszczególnych elementów, warto poddać się prostemu eksperymentowi poznawczemu – przyjrzeć się kilku przykładom kodu, następnie upewnić się, jaki osiągają efekt i dopiero później zapoznać się ze szczegółami wyjaśniającymi działanie poszczególnych elementów. Dzięki temu czytelnik dysponujący ponadpodstawowym poziomem wiedzy o programowaniu będzie mógł wyrobić sobie szczerszą opinię na temat, między innymi, czytelności prezentowanych narzędzi.

Biorąc pod uwagę nacisk kładziony na odpowiednie grupowanie narzędzi w bibliotece standardowej C++, widoki i zakresy zostały objęte nowymi przestrzeniami nazw. Dlatego aby utrzymać zwięzłość przykładów, zakłada się, że listingi przedstawione w dalszej części tego rozdziału są wzbogacone w aliasy przestrzeni nazw zaprezentowane w Listing 61.

Listing 61: Aliasy przestrzeni nazw `std::views` oraz `std::ranges`.

```
1  namespace vs = std::views;
2  namespace rg = std::ranges;
```

Na początek warto przyjrzeć się czwartemu sposobowi na osiągnięcie tego samego, co poszczególne wywołania `std::ranges::sort()` z Listing 43. Został on zaprezentowany w Listing 62.

Listing 62: Sortowanie pierwszych trzech elementów wektora przy pomocy sprecyzowania podzakresu dzięki adapterom.

```
1 auto main() -> int {
2     auto numbers = std::vector<int>{5, 4, 3, 2, 1};
3     rg::sort(numbers | vs::take(3));
4 }
```

Jeżeli celem byłoby posortować nie pierwsze trzy elementy, tylko trzy środkowe (zakładając wielkość wektora równą 5), to można skorzystać z adaptera `drop`.

Listing 63: Sortowanie trzech elementów z pominięciem pierwszego przy pomocy widoków `drop` oraz `take`.

```
1 auto main() -> int {
2     auto numbers = std::vector<int>{5, 4, 3, 2, 1};
3     rg::sort(numbers | vs::drop(1) | vs::take(3));
4 }
```

Jak można zauważyć w Listing 63, operator *pipe* może być kilkukrotnie użyty do *komponowania* wielu adapterów w jeden widok. Zawartość wektora `numbers` po wykonaniu algorytmu sortowania z omawianego listingu to: `5 2 3 4 1`. Jak widać, środkowe trzy elementy zostały posortowane.

Kolejnym dobrym przykładem może być translacja kodu z Listing 57 (powtórzony dla wygody w Listing 65), gdzie dzięki funkcyjnemu podejściu można było w elegancki sposób wygenerować odpowiednie potęgi liczby 2.

Listing 64: Funkcyjne rozwiązanie problemu generowania wybranych potęg dwójki w C++20.

```
1 auto powersOfTwo = vs::iota(0)
2         | vs::filter([](int const i) { return i % 2 == 0; })
3         | vs::transform([](int const i) { return std::pow(2, i); })
4         | vs::take(10);
5
6 for (auto const n : powersOfTwo) {
7     std::cout << n << ' ';
8 }
```

Warto porównać te dwa listingi (Listing 64 i Listing 57) w celu zidentyfikowania podobieństw i różnic oraz wad i zalet implementacji narzędzi programowania funkcyjnego z obydwu języków. W tym celu Listing 57 został powtórzony w Listing 65, aby ułatwić wizualną analizę.

Listing 65: Powtórzony Listing 57.

```
1  IntStream.iterate(0, i -> i + 1)
2          .filter(i -> i % 2 == 0)
3          .map(i -> (int)Math.pow(2, i))
4          .limit(10)
5          .forEach(System.out::println);
```

W ramach wizualnych różnic nie sposób jest nie dostrzec trzech podstawowych, różniących się notacji. Różnice te zostały ze sobą zestawione w Tabela 2.

Tabela 2: Porównanie konstrukcji programowania funkcyjnego pod względem aspektów wizualnych z C++20 oraz Javy 8.

Widoki z C++20	Strumienie z Javy 8
Widoki wymagają pętli (lub standardowego algorytmu) do iteracji po zakresie, który reprezentują.	Strumienie wspierają operacje końcowe (ang. *terminal operation*), które zastępują pętle i algorytmy, dzięki czemu pozwalają na kompozycję tworzonej struktury funkcyjnej wraz z jej konsumpcją.
Korzystają z operatora *pipe* do łączenia z arbitralnymi, innymi adapterami i widokami.	Korzystają z operatora kropki, która ogranicza się wyłącznie do wykorzystywania metod zdefiniowanych na obiekcie reprezentującym konstrukcję funkcyjną.
Notacja lambd jest bardzo rozwlekła (ang. *verbose*).	Notacja lambd jest zwięzła.

Porównanie zwięzłości składni lambd w Tabela 2 może się wydawać nieadekwatne do omawianego tematu, ale z uwagi na fakt, że są one praktycznie nieodłącznym elementem programowania funkcyjnego przy użyciu strumieni z Javy 8 czy widoków z C++20, takie porównanie jest jak najbardziej merytoryczne.

Poza aspektami wizualnymi warto również porównać aspekty funkcjonalne, które zostały zestawione w Tabela 3.

Tabela 3: Porównanie konstrukcji programowania funkcyjnego pod względem aspektów funkcjonalnych z C++20 oraz Javy 8.

Widoki z C++20	Strumienie z Javy 8
Zdefiniowane jako „wolne obiekty". Nowe adaptery mogą być dowolnie łączone ze starymi.	Narzędzia zdefiniowane jako metody. Rozszerzalność jest wspierana przy pomocy dziedziczenia i dostarczenia nowych metod – brak możliwości wzbogacenia „starych konstrukcji"

	o nowe funkcjonalności bez zmiany typu / wersji języka.
Z założenia gotowe do użytku wiele razy. Pętlę zaprezentowaną w Listing 64 można wykonać wielokrotnie na obiekcie `powersOfTwo`. Wyjątki stanowią wyłącznie konstrukcje, które z założenia reprezentują nieodwracalnie zmienny stan (np. widok na elementy pobierane z pliku lub z sieci).	Z założenia przystosowane do jednorazowego użytku. Zapisanie strumienia do zmiennej i późniejsze wykonanie operacji końcowej sprawia, że nie można wykonać tego typu operacji na tej samej zmiennej ponownie.

Ostatnim przykładem prezentującym wygodę i ekspresywność widoków i zakresów będzie ulepszona wersja Listing 41, który weryfikował, czy przekazany tekst był palindromem.

Listing 66: Implementacja algorytmu z Listing 41 z wykorzystaniem zakresów i widoków[25].

```
1  auto isPalindrome(std::string const& str) -> bool {
2      return rg::equal(str, str | vs::reverse);
3  }
```

Na koniec warto zauważyć, że nie każdy zakres jest widokiem. Przykładem jest `std::vector<T>`. O ile spełnia on Koncept `std::ranges::range` oraz `std::movable`, to specjalizacja `enable_view` dla niego ma wartość `false` (domyślną). Jest tak dlatego, żeby móc kontrolować, które elementy wchodzą w interakcję z widokami oraz ułatwić rozszerzanie funkcjonalności adapterów, które mogą zakładać, że otrzymają do adaptowania wyłącznie widoki.

Przykład, problemu, w którym powyższe założenie ma znaczenie, został zaprezentowany w rozdziale **6.3.4**.

Jeżeli lewy operand operatora *pipe* będzie zakresem, ale nie widok, to *pipe* domyślnie go opakuje w typ reprezentujący widok. Dzięki tym dwóm narzędziom (odróżnienie widoku od zakresu oraz możliwość wygodnego stworzenia widoku z zakresu), programowanie z użyciem adapterów jest bardzo wygodne.

Na koniec warto jeszcze zwrócić uwagę na bardzo istotną korelację między widokami a ideą biblioteki Zakresów. Aby dany adapter mógł być zaaplikowany do zakresu, to zakres ten nie może być reprezentowany przez parę iteratorów (jako oddzielne zmienne). Koniec końców i tak należałoby wpierw je pogrupować w jeden typ, który byłby później adaptowany. Jest to kolejny powód, dla którego biblioteka

[25] Z perspektywy czysto wydajnościowej, ten kod może skutkować wygenerowaniem gorszego kodu maszynowego niż wersja zaprezentowana w Listing 41. To dlatego, że po pierwsze widok `reverse` jest nieco bardziej złożony niż „zwykły" iterator odwracający. Dodatkowo, algorytm ten przejdzie po całym zakresie, a nie tylko porówna obydwie jego połówki. Pierwszy problem ewidentnie dotyczy klasycznego wyboru między optymalnością a czytelnością, dlatego autor pracy nie uznaje go jako kluczowy. Dodatkowo, kompilator może wyprodukować identyczny kod dla obydwu wersji. Drugi problem można naprawić adaptując obydwa zakresy za pomocą `vs::take(str.size() / 2)`, ale zostało to pominięte w celu zwiększenia przejrzystości przykładu.

standardowa ewoluowała w kierunku reprezentacji zakresów jako pojedyncze obiekty, nawet jeżeli ta reprezentacja to nic innego jak tylko opakowanie na parę iteratorów.

5 Pomniejsze usprawnienia narzędzi języka C++

Poniższe podrozdziały omawiają wybrane usprawnienia, które aplikują się do języka C++ lub jego ekosystemu. Wspólną ich cechą jest to, że swoje istnienie zawdzięczają nowym elementom dodanym w C++20 – głównie Konceptom. Zapoznanie się z nimi pozwala zauważyć pewne nieoczywiste, pozytywne konsekwencje usprawniania działania elementów języków programowania.

5.1 Uproszczenie wybranych konstrukcji programistycznych

W Listing 19 zostało zaprezentowane użycie szablonu `std::swap()`. Algorytm ten wykorzystuje prostą technikę zamiany, która przenosi[26] lewy parametr do lokalnej zmiennej, następnie go nadpisuje przeniesieniem prawego, a na koniec nadpisuje prawy operand przenosząc wartość wspomnianej zmiennej lokalnej.

Algorytm ten ma niestety pewną wadę – jeżeli istnieje wydajniejszy sposób niż przenosząca konstrukcja tymczasowej zmiennej na zamianę wartościami dwóch obiektów danego typu, to jest on nieoptymalny. Stąd programiści byli zachęcani do definiowania optymalnych dla tworzonych przez siebie typów mechanizmów zamiany, np. w postaci wolnej funkcji `swap()`, metody `.swap()` lub specjalizacji szablonu `std::swap()`[27].

Skutkowało to jednak nowymi problemami:

- Którego algorytmu użyć? Czasem nieoptymalnego `std::swap()`?
- Skąd wiadomo, czy istnieje specjalizacja `std::swap()`?
- Co jeżeli zamiast specjalizacji została zdefiniowana wolna funkcja lub metoda o tej nazwie?
- Jak zachowa się specjalizacja w przypadku hierarchii dziedziczenia i wysyłania obiektów różnego typu?

Wszystkie te problemy doprowadziły do powstania wzorca o nazwie *dwustopniowa zamiana* (ang. *two-step swap idiom*) [42] [43]. Polega on na tym, że przed umieszczeniem wywołania `swap()` umieszczana jest deklaracja `using std::swap;`, dzięki której ADL (ang. *argument-dependent lookup*) może wybrać odpowiednią implementację.

Dzięki Konceptom można nieco ułatwić ten proces korzystając z serii wyrażeń `requires()`, które zwyczajnie by sprawdzały, która wersja algorytmu istnieje, a następnie ją wywołać.

[26] W tym kontekście termin *przeniesienie* dotyczy semantyk przenoszenia (ang. *move semantics*), które zostały wspomniane w rozdziale **3.3**.

[27] O ile dodawanie nowych elementów i definicji do przestrzeni nazw std skutkuje zachowaniem niezdefiniowanym (UB), to dodawanie specjalizacji szablonowych stanowi jeden z nielicznych wyjątków od tej reguły.

Na szczęście nie ma takiej potrzebny, ponieważ dokładnie to robi narzędzie `std::ranges::swap()`. Tak samo jak nie istnieje powód korzystania z `std::sort()` zamiast `std::ranges::sort()`, to nie istnieje powód korzystania z `std::swap()` zamiast z `std::ranges::swap()`.

Kolejnym uproszczeniem jest użycie klauzuli `requires` do warunkowego definiowania metod w klasach. Przykład ten jest dokładnie opisany w [44]. Można go nieco uprościć do pseudokodu zaprezentowanego w Listing 67.

Listing 67: Warunkowe deklarowanie i definiowanie destruktora dzięki Konceptom.

```
1  template <typename T>
2  concept ObjectNeedsCustomDestruction = …;
3
4  template <typename T>
5  struct Container {
6      T object;
7
8      ~Container() requires ObjectNeedsCustomDestruction<T>{
9          destroyInnerObject(object);
10     }
11 };
```

Alternatywą dla danego rozwiązania (oraz warunkowego dostarczenia potencjalnie kilku wersji destruktorów) było warunkowe dziedziczenie z wybranej klasy bazowej, np. dzięki szablonowi `std::conditional_t` połączonemu z techniką CRTP (ang. *curiously recurring template pattern*).

5.2 Usprawnienia środowisk programistycznych

Programowanie generyczne cechuje się, między innymi, brakiem dokładnej znajomości typu danych, na których obecnie tworzone narzędzie będzie pracowało. Stąd płynie jego generyczność – takie narzędzie powinno współpracować z każdym typem, który spełnia jakieś warunki.

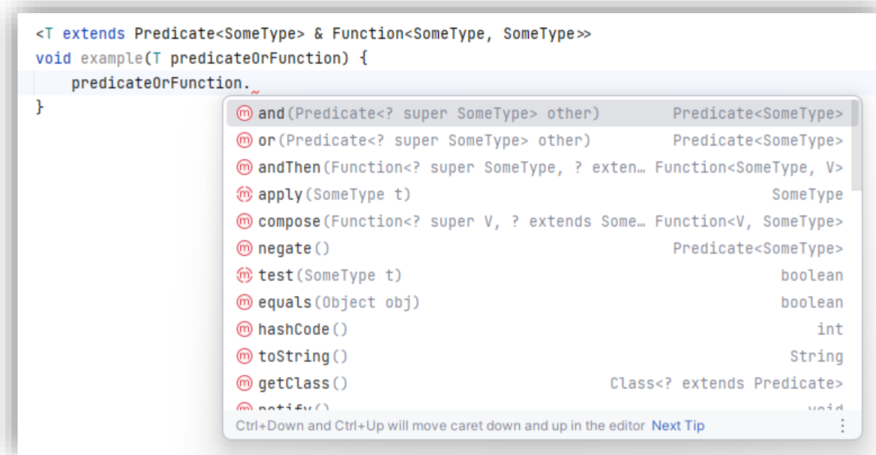
Warto w tym momencie zwrócić szczególną uwagę na końcówkę ostatniego zdania. Traktuje ona o tym, że algorytm (lub struktura danych) **oczekuje** od typu wspierania jakiś dobrze zdefiniowanych zachowań, np. udostępnienie implementacji jakiejś metody. Oznacza to, że projektując generyczne narzędzie i definiując te wymagane zachowania, *coś* jest jednak wiadome odnośnie typów, z którymi wspomniane narzędzie będzie współpracowało. Można to łatwo zaobserwować pisząc generyczną metodę w języku Java, która ogranicza typ wymogiem implementowania danego interfejsu, co zostało zaprezentowane w Listing 68.

Listing 68: Generyczna metoda przyjmująca dowolny typ, który implementuje dwa sprecyzowane interfejsy w Javie.

```
1  <T extends Predicate<SomeType> & Function<SomeType, SomeType>>
2  void example(T predicateOrFunction) {
       predicateOrFunction.
```

```
3  }
4
```

Linijka 3 z Listing 68 została celowo pozbawiona jakiejkolwiek nazwy po kropce, w celu zaprezentowania wygodnych podpowiedzi środowisk programistycznych, które zostały zawarte w Rysunek 4.



Rysunek 4: Lista dostępnych funkcjonalności podpowiadanych przez środowisko JetBrains IntelliJ IDEA dla obiektu typu generycznego w Javie.

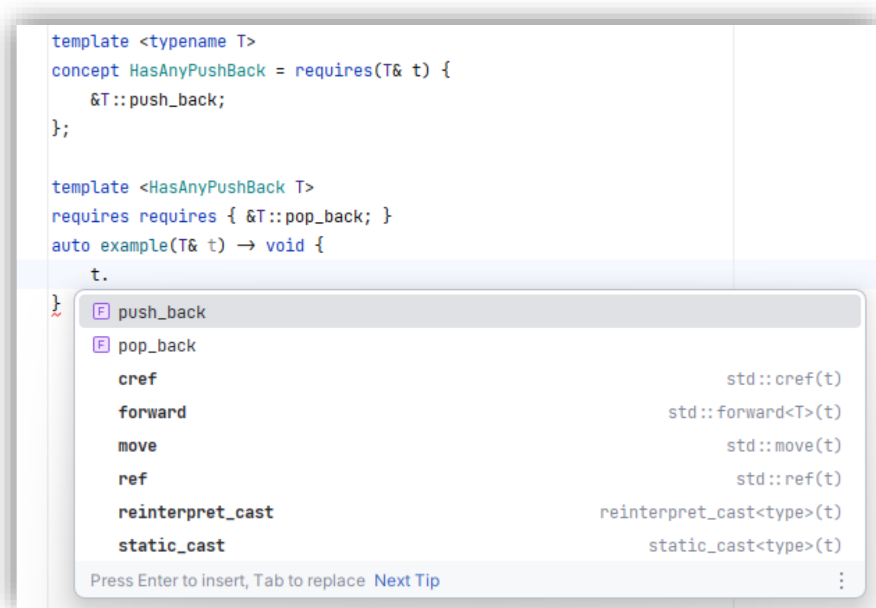
Jest to niewątpliwie przydatna funkcjonalność, zwłaszcza z uwagi na stopień zaawansowania aktualnych środowisk programistycznych.

W przypadku korzystania z „surowych" szablonów w C++, żadne środowisko nie podpowie możliwych do użycia metod (być może w ogóle ich nie ma, bo dany typ jest używany z prymitywem, np. `int`). Sytuacja jest na szczęście inna dzięki Konceptom. Wprowadziły one dodatkowe metainformacje dotyczące pisanego kodu, dzięki czemu takie środowiska jak JetBrains CLion mogą podpowiadać dostępne na danym typie generycznym metody bazując na sprecyzowanych wymaganiach w klauzuli `requires`. Dla kodu zaprezentowanego w Listing 69 (z wybrakowaną linią numer 9), środowisko CLion podpowiada kontekstowe akcje zaprezentowane w Rysunek 5.

Listing 69: Generyczna metoda przyjmująca dowolny typ, który spełnia sprecyzowany Koncept oraz lokalne wymaganie w C++.

```
1  template <typename T>
2  concept HasAnyPushBack = requires(T& t) {
3      &T::push_back;
4  };
5
6  template <HasAnyPushBack T>
7  requires requires { &T::pop_back; }
8  auto example(T& t) -> void {
9
```

```
10      t.
    }
```



Rysunek 5: Lista dostępnych funkcjonalności podpowiadanych przez środowisko JetBrains CLion dla obiektu typu generycznego w C++.

6 Implementacja rozszerzenia kontenerów biblioteki standardowej C++

Poniższe rozdziały przedstawiają proces identyfikacji niedoskonałości w bibliotece standardowej języka C++20. W wyniku tego procesu powstał również plan implementacji udoskonalenia tej biblioteki, który również został szczegółowo opisany. Fragmenty wynikowej implementacji zostały przedstawione i omówione, wraz z porównaniem ich do alternatywnych rozwiązań.

6.1 Niekompletna natura biblioteki standardowej języka C++

Istotnym problemem procesu standaryzacji języka C++ jest dyskusja, poprawki i akceptowanie lub odrzucanie wniosków standaryzacyjnych (ang. *proposals*). Natura tego problemu polega na tym, że jest to bardzo kosztowny zasobowo proces. Wymaga bardzo dużo czasu, staranności i niekiedy szczęścia w kontekście akurat obecnych osób podczas zgromadzenia komitetu, który decyduje, czy dany wniosek jest aplikowany do standardu.

Oczywiście istnieją też plusy tego rozwiązania – standaryzacja w tej formie wymusza proces bardzo skrupulatnej analizy każdej zmiany. Dodatkowo, uniemożliwia jednemu ciału (np. firmie) zdobycie monopolu na rozwój narzędzia. C++, jako jeden z najpopularniejszych języków programowania, w którym jest napisana i utrzymywana ogromna liczba aplikacji, jest elementem, na którym polega bardzo wiele jednostek. Nagła i nieprzemyślana zmiana kierowana celem jednej firmy mogłaby być katastrofalna w skutkach dla innych firm.

Niestety, proces standaryzacji jest z reguły bardzo wolny. Oznacza to, że nowe narzędzia, które są relatywnie duże, zwykle są dodawane do języka w wybrakowanej formie. Nie inaczej jest z biblioteką Zakresów. Rozdział **4.1** dokładnie opisał jej ideę i powód, dla którego warto reprezentować koncepcję tej abstrakcji jako pojedynczy obiekt zamiast jako para iteratorów. Aby zobrazować niekompletność tej biblioteki, warto przyjrzeć się prostemu przykładowi usuwania duplikatów z wektora, który został zaprezentowany w Listing 70.

Listing 70: Usuwanie duplikatów z `std::vector`.

```
1  auto main() -> int {
2      auto numbers = std::vector<int>{
3          1, 2, 3, 1, 2, 3,
4          3, 2, 1, 3, 2, 1,
5          1, 1, 2, 2, 3, 3
6      };
7
8      std::ranges::sort(numbers);
9      auto duplicates = std::ranges::unique(numbers);
10     numbers.erase(duplicates.begin(),duplicates.end());
11 }
```

12

Od razu można zauważyć pewną niekonsekwentność. Zakresy miały na celu modernizację interfejsu pracy na zakresach danych pod postacią reprezentacji tej abstrakcji jako pojedynczego obiektu. Taką zmianę można zaobserwować między innymi w typie zwracanym przez `std::ranges::unique()`. `std::unique()` zwraca pojedynczy iterator na początek zakresu, który użytkownik powinien usunąć. `std::ranges::unique()` zwraca widok[28] na cały (pod)zakres do usunięcia. Jest to bardzo wygodna zmiana. Mimo niej, i tak trzeba od razu wypakować te dwa iteratory, aby skorzystać z metody `.erase()`. Znacznie wygodniej byłoby skorzystać z notacji `numbers.erase(duplicates)`.

Dzięki wprowadzeniu Konceptów i Zakresów, taka implementacja jest relatywnie prosta. Pozostała część tej sekcji pracy została poświęcona identyfikacji elementów, które nie doczekały się uzakresowionych operacji w kontekście kolekcji biblioteki standardowej języka C++, a następnie uzupełnieniu ich o te elementy.

6.2 Identyfikacja elementów kolekcji biblioteki standardowej, które można wzbogacić o uzakresowione konstrukcje

W tym podrozdziale termin „kolekcja" odnosić się będzie do zakresu, który jest właścicielem swoich elementów. Kolekcjami przykładowo są:

- `std::vector`;
- `std::array`;
- `std::string`;
- `std::stack`;
- `std::map`;
- `std::list`.

Ale kolekcjami już nie są:

- `std::span`;
- widoki ani adaptery (`std::views`);
- typ `std::ranges::subrange`.

Warto też zwrócić uwagę, że o ile krotka (`std::tuple`) jest właścicielem swoich obiektów, to nie jest uznawana za kolekcję.

[28] Tutaj termin „widok" nie został użyty w kontekście narzędzia `std::views`.

Kolekcje w C++ są podzielone zgodnie z następującą hierarchią:

- **Kontenery sekwencyjne**, do których należą:
  - `std::deque`;
  - `std::vector`;
  - `std::list`;
  - `std::forward_list`.
- **Kontenery asocjacyjne**, do których należą:
  - `std::map`;
  - `std::unordered_map`;
  - `std::multimap`;
  - `std::unordered_multimap`;
  - `std::set`;
  - `std::unordered_set`;
  - `std::multiset`;
  - `std::unordered_multiset`.
- **Adaptery kontenerów**, do których należą:
  - `std::stack`;
  - `std::queue`;
  - `std::priority_queue`.

W momencie pisania pracy, standard C++23 został już sfinalizowany. Praca nie koncentruje się na elementach dodanych w nim, ale warto wspomnieć, że pierwszy raz od długiego czasu[29] biblioteka kontenerów dostała urozmaicona o nowe adaptery kontenerów:

- `std::flat_set`;
- `std::flat_map`;
- `std::flat_multiset`;
- `std::flat_multimap`.

[29] Ostatnio dodanymi kontenerami były `std::array` oraz kontenery nieuporządkowane (ang. *unordered*) z C++11.

Na skutek analizy elementów tych kolekcji, które można wzbogacić o uzakresowione narzędzia, powstała implementacja rozszerzająca funkcjonalność standardowych kontenerów. Dzięki niej, kod z Listing 70 można zastąpić kodem z Listing 71.

Listing 71: Usuwanie duplikatów z `masters::vector`.

```
1  #include <algorithm>
2
3  #include <s17137/sequential/vector.hpp>
4
5  auto main() -> int {
6      auto numbers = masters::vector<int>{
7          1, 2, 3, 1, 2, 3,
8          3, 2, 1, 3, 2, 1,
9          1, 1, 2, 2, 3, 3
10     };
11
12     std::ranges::sort(numbers);
13     auto duplicates = std::ranges::unique(numbers);
14     numbers.erase(duplicates);  // rangified
15 }
```

Te dwa, omawiane listingi różnią się efektywnie tylko w dwóch miejscach. Po pierwsze, typ `std::vector` został zamieniony na `masters::vector`, który wspiera wersję metody `.erase()` przyjmującą podzakres, zamiast pary iteratorów (druga zmiana).

Metoda `.erase()` to jedynie jeden z wielu elementów interfejsu kolekcji z języka C++, które warto uzakresowić. Kompletna lista tych elementów, która została wzięta pod uwagę w pracy, prezentuje się następująco:

1. konstruktory oraz metoda `.assign()`;
2. metody `.insert()` oraz `.insert_after()`;
3. metody `.erase()` oraz `.erase_after()`;
4. metody `.splice()` oraz `.splice_after()`;
5. metody `.append()` oraz `.replace()`.

Warto zwrócić uwagę, że metody z `_after()` w nazwie są unikalne dla typu `std::forward_list`, a `.append()` oraz `.replace()` są unikalne dla `std::string`.

Ze względu na obraną strategię rozszerzenia funkcjonalności kolekcji, która została omówiona w następnym rozdziale **6.3**, należy również zwrócić uwagę na konieczność zaimplementowania:

- Operatorów porównujących (ang. *comparison operators*). Na szczęście, dzięki wprowadzonemu w C++20 operatorowi <=>, okazało się być to relatywnie proste.
- Wskazówek dedukcji argumentów szablonowych dla klas (ang. *CTAD – class template argument deduction guides*).

Omówienie implementacji i decyzji argumentujących dane podejście zostało przedstawione w dalszej części tej sekcji pracy.

6.3 Implementacja elementów kolekcji biblioteki standardowej, które można wzbogacić o uzakresowione konstrukcje

Istnieje kilka sposobów na zmianę zachowania elementów ze standardowej biblioteki języka C++. W przypadku kolekcji są to:

1. własne implementacje biblioteki standardowej;
2. tworzenie typów dziedziczących po ww. kolekcjach;
3. własne, niezależne implementacje.

Przewagi punktu 1 dotyczą integralności stworzonego efektu. Mając możliwość edytowania kodu biblioteki standardowej uzyskuje się „najprawdziwszy" efekt testowania jej modyfikacji. Dodatkowo, całe środowisko wspierające proces wytwarzania tej biblioteki stanowi automatyczne wsparcie tej modyfikacji. Mowa tutaj o, między innymi, przygotowanych testach przez różnych dystrybutorów implementacji bibliotek standardowych. Mówiąc krótko, edytując kod źródłowy gotowej biblioteki standardowej (np. Microsoftu, libc++ czy libstdc++) „automatycznie" można skorzystać z oficjalnego zestawu testów, który jest do niego przygotowany.

Minusem tego rozwiązania jest ogrom pracy, który trzeba włożyć w projekt. Modyfikacje biblioteki standardowej nie są trywialne. Jej szczegóły implementacyjne nie są wytwarzane z myślą przede wszystkim o czytelności i łatwej, późniejszej zmianie, jak to bywa (lub przynajmniej powinno być) w przypadku kodu biznesowego. Dodatkowo, z uwagi na charakterystykę szablonów i na fakt, że edytowane elementy są szablonami, nie można skorzystać z wygodnej separacji implementacji. Szablony nie mogą zostać ukryte za interfejsem z uwagi na strategię ich kompilacji. Oznacza to, że każda modyfikacja kodu szablonu będzie skutkowała rekompilacją każdego pliku, który dołącza tę implementację.

Przewagą punktu 2 jest prostota i wygoda implementacyjna. Dziedziczenie jest narzędziem stworzonym do rozszerzania funkcjonalności. W teorii wystarczy jedynie publicznie dziedziczyć z danych kontenerów i dopisać odpowiednie funkcjonalności.

Niestety, publiczne dziedziczenie w C++ potrafi powodować trudny do identyfikacji problem. Publiczne dziedziczenie sugeruje wykorzystanie polimorficzne. Nietrudno sobie wyobrazić sytuację, gdzie użytkownik tworzy wskaźnik do `std::vector<T>`, który będzie wskazywał na dynamicznie alokowany obiekt typu `masters::vector<T>`. Na pierwszy rzut oka ta sytuacja nie wygląda na problematyczną. Niestety, próba zwolnienia takiej pamięci przy pomocy wspomnianego wskaźnika zakończy się wywołaniem zachowania niezdefiniowanego (UB). To dlatego, że destruktor szablonu `std::vector<T>` nie jest wirtualny.

Jest to poważny problem. Jak już zostało przytoczone, dziedziczenie publiczne sugeruje polimorficzną hierarchię, której częstym przypadkiem użycia jest posiadanie wskaźników typu bazowego do

dynamicznie alokowanych obiektów dziedziczących. Pamięć zaalokowana dynamicznie winna być ręcznie zwalniana (obejmując również przypadek wykorzystania RAII). Jeżeli taką pamięć zwolni się używając wskaźnika na typ bazowy, który nie ma oznaczonego destruktora jako wirtualnego, wywołanie programu może skutkować przeróżnymi, niepożądanymi efektami.

Przewagą punktu nr 3 jest pełna kontrola nad narzędziem. Tworząc kontenery „od zera" unika się problemów dotyczących poprzednio omawianych punktów. Niestety, traci się również ich zalety. Nie ma się dostępu do ekosystemu wspierającego bibliotekę standardową i nie można zautomatyzować większości implementacji (jak np. przy dziedziczeniu).

Autor niniejszej pracy zdecydował się na strategię nr 2. To dlatego, że jej wadę można relatywnie prosto zniwelować oraz dlatego, że przewagi innych rozwiązań były nieproporcjonalne do ich wad.

6.3.1 Rozwiązanie problemu braku wirtualnego destruktora w typach bazowych za pomocą prywatnego dziedziczenia

Przytoczony wcześniej problem błędnego (polimorficznego) użycia nowozdefiniowanego typu można łatwo rozwiązać zastępując publiczne dziedziczenie prywatnym. Dzięki temu nowy typ danych odziedziczy wszystkie pożądane funkcjonalności. Jednocześnie, każda próba zdefiniowania wskaźnika lub referencji typu bazowego do obiektu o omawianym typie będzie skutkować błędem kompilacji. Początek implementacji został zaprezentowany w Listing 72.

Listing 72: Deklaracja i definicja szablonu `masters::vector`.

```
1  namespace masters {
2      template <typename T, typename Allocator = std::allocator<T>>
3      class vector : std::vector<T, Allocator> { };
4  }
```

W języku C++, typ wprowadzany przez słowo kluczowe `class` cechuje się domyślnym specyfikatorem dostępu `private`. Tyczy się to również dziedziczenia, stąd w omawianym przykładzie nowy szablon `masters::vector` dziedziczy prywatnie po `std::vector<T, Allocator>`.

Należy pamiętać o tym, że szablon `std::vector` posiada nie jeden, a dwa argumenty szablonowe. Pierwszy określa typ przechowywanego elementu, a drugi alokator. Ta uwaga aplikuje się do każdego, nowego szablonu klas stworzonego w ramach przestrzeni nazw `masters`.

Jest to poprawne podejście, lecz generuje nowy problem. Wprowadza zależne nazwy (ang. *dependent name*). Został on omówiony w dalszej części pracy.

6.3.2 Problem widoczności odziedziczonych elementów w kontekście dziedziczenia prywatnego

Dziedziczenie prywatne rozwiązało problem błędnej sugestii zachęcającej do polimorficznego użycia szablonu `masters::vector`, ale jednocześnie wprowadziło kilka dodatkowych problemów. Jednym z

nich jest widoczność odziedziczonych pól. Są one również prywatne. Przykład tego problemu obrazuje Listing 73.

Listing 73: Brak możliwości dostępu do publicznego pola, które zostało odziedziczone prywatnie.

```
1  struct Base {
2      int publicField;
3  };
4
5  struct Derived : private Base { };
6
7  auto useField(Derived derived) -> void {
8      derived.publicField;    // compilation error: publicField is private
                                  in this context
9  }
```

Nawet jeżeli dane pole (czy to obiekt, czy to metoda, czy to alias) jest w danej klasie publiczne, to prywatne dziedziczenie jednoznacznie określa, że tylko klasa dziedzicząca „wie" o tym, że dziedziczy z klasy bazowej. Inne byty, jak np. funkcja `useField()`, nie wie o tej relacji. Oznacza to, że nie wie iż `Derived` odziedziczyło `publicField` po `Base`.

Rozwiązaniem tego problemu jest upublicznienie pola. Można to w prosty sposób zrealizować za pomocą deklaracji `using`, jak zostało to przedstawione w Listing 74.

Listing 74: Przykład użycia deklaracji `using` w celu upublicznienia prywatnie odziedziczonego pola.

```
1  struct Base {
2      int publicField;
3  };
4
5  struct Derived : private Base {
6      using Base::publicField;    // change here
7  };
8
9  auto useField(Derived derived) -> void {
10     derived.publicField;        // OK
11 }
```

Warto zwrócić uwagę, że ten mechanizm pozwalałby na upublicznienie pola `publicField` nawet gdyby było one oznaczone jako `protected`. Nie zadziałałoby to jednak w przypadku, gdyby pole to było prywatne. To dlatego, że niezależnie od hermetyzacji dziedziczenia, pole prywatne jest widoczne tylko dla klasy, w której się znajduje[30], więc `Derived` nie wiedziałoby w ogóle o istnieniu takiego pola.

Wydaje się, że rozwiązanie problemu jest proste. Wbrew pozorom tak właśnie jest. Należy zwyczajnie opatrzeć deklaracją `using` wszystkie elementy, które są domyślnie publiczne w `std::vector`.

[30] Wyjątkiem od tej reguły jest odpowiednio użyty modyfikator `friend`, który nie ma w tym przykładzie znaczenia.

Niestety tych elementów jest bardzo dużo. Sam `std::vector` liczy takich pól aż 43. Są to, między innymi:

- konstruktory (wszystkie można na raz upublicznić jedną dyrektywą);
- aliasy aplikujące się dla każdego kontenera takie jak `value_type`, `reference`, `const_reference`, `iterator`, `const_iterator`, itd.;
- metody `.begin()`, `.end()`, `.swap()`, `.size()`, `.empty()` i inne, które są wspólne dla każdego kontenera;
- metody `.rbegin()`, `.crbegin()`, `.reserve()`, `.clear()`, `.emplace()`, itd.

Aby ułatwić implementację i uniknąć niepotrzebnej duplikacji kodu, autor pracy zdecydował się na wykorzystanie mechanizmu makr omówionego w rozdziale **3.1**. Powstało, między innymi, makro zaprezentowane w Listing 75.

Listing 75: Definicja makra `GENERATE_CONTAINER_REQUIREMENTS_TYPES_FROM()` automatyzującego generowanie kodu upubliczniającego wszystkie pola będące wspólne dla każdego kontenera.

```
1  #define GENERATE_CONTAINER_REQUIREMENTS_TYPES_FROM(source_name) \
2  using value_type = typename source_name::value_type; \
3  using reference = typename source_name::reference; \
4  using const_reference = typename source_name::const_reference; \
5  using iterator = typename source_name::iterator; \
6  using const_iterator = typename source_name::const_iterator; \
7  using difference_type = typename source_name::difference_type; \
8  using size_type = typename source_name::size_type; \
9  using source_name::source_name; \
10 using source_name::operator=; \
11 using source_name::begin; \
12 using source_name::end; \
13 using source_name::cbegin; \
14 using source_name::cend; \
15 using source_name::swap; \
16 using source_name::size; \
17 using source_name::max_size; \
18 using source_name::empty;
```

Użycie tego makra jest bardzo proste. Listing 72 wystarczy zmodyfikować w sposób pokazany w Listing 76.

Listing 76: Szablon `masters::vector` wzbogacony o upublicznienie wspólnych dla wszystkich kontenerów pól.

```
1  namespace masters {
2      template <typename T, typename Allocator = std::allocator<T>>
3      class vector : std::vector<T, Allocator> {
4          using base = std::vector<T, Allocator>;
```

```
5        public:
6            GENERATE_CONTAINER_REQUIREMENTS_TYPES_FROM(base)
7        };
8    }
```

Widoczna deklaracja `using base = std::vector<T, Allocator>;` ma dwa zastosowania. Po pierwsze zwiększa czytelność użycia makra `GENERATE_CONTAINER_REQUIREMENTS_TY PES_FROM()`. Po drugie pozwala na łatwiejsze rozwiązanie problemu nazw zależnych, który został wspomniany na końcu rozdziału **6.3.1** [45].

Pozostałe (unikalne dla kontenerów sekwencyjnych lub dla samego typu `std::vector`) pola zostały upublicznione w podobny sposób – albo za pomocą pomocniczych makr, albo za pomocą lokalnych deklaracji `using`.

6.3.3 Definicja szablonów metod akceptujących zakres wartości w postaci pojedynczego obiektu

Listing 71 prezentuje możliwość wywołania metody `.erase()` usuwającej zakres elementów należących do danego wektora. W tym przykładzie została zaprezentowana wersja metody, w której zakres jest precyzowany przez pojedynczy obiekt. Tutaj jest to typ zwracany z wywołania algorytmu `std::ranges::unique()`, który z kolei zwróciło typ `std::ranges::subrange<Iterator>`, gdzie `Iterator` to typ iteratora do przekazanego kontenera.

Implementacja metody `.erase()` może zatem być bardzo prosta. Zaprezentowana została w Listing 77.

Listing 77: Implementacja uzakresowionej wersji metody `.erase()`.

```
1    auto erase(
2            std::ranges::subrange<const_iterator> const& rng
3    ) -> auto {
4        return erase(std::ranges::begin(rng), std::ranges::end(rng));
5    }
```

`const_iterator` to alias obecny w każdym kontenerze. Szablon `std::ranges::subrange` ze sprecyzowanym typem iteratora na wspomniany stały iterator pozwoli na stworzenie widoku na dowolny podzakres danej kolekcji. Dodatkowo, szablon ten wspiera konstruktory konwertujące. To bardzo istotne, ponieważ, przykładowo, `std::ranges::unique()` zwykle zwraca `std::ranges::subrange<iterator>`, a nie `std::ranges::subrange<const_iterator>`. To dlatego, że algorytm ten potrzebuje działać na iteratorach pozwalających na edycję danych. Na szczęście podzakres „stały” jest bardziej restrykcyjny niż podzakres pozwalający na modyfikację danych, więc na podstawie drugiego można niejawnie stworzyć pierwszy.

Implementacja tej metody to prosta delegacja do odziedziczonego, oryginalnego narzędzia, czyli to „klasycznej” wersji metody `.erase()`. To trend, który powtarza się przy wielu takich implementacjach.

Odwrotną operacją do usuwania danych jest ich dodawanie. `.insert()` pozwala na dodawanie danych, ale nie powinien on akceptować takiego samego argumentu jak `.erase()`. Powód obrazuje Listing 78.

Listing 78: Przykład dodania elementów z kontenera `std::string` do kontenera `std::vector`.

```
1  auto main() -> int {
2      auto string = std::string("example");
3      auto vector = std::vector<char>();
4
5      vector.insert(
6              vector.end(),
7              string.begin(), string.end()
8      );
9  }
```

Przykład jest bardzo prosty. Do wektora znaków dodawane są elementy ciągu tekstowego. Typ elementów się zgadza, ale typ iteratorów nie. Nie powinno to jednak stanowić żadnej przeszkody w wykonaniu takiej operacji. Co więcej, warto również zezwolić na kopiowanie elementów, które można niejawnie przekonwertować na docelowy typ, co zostało zaprezentowane w Listing 79.

Listing 79: Przykład dodania elementów z jednego wektora do drugiego, gdzie elementy są różnego typu, ale wspierają niejawną konwersję.

```
1  auto main() -> int {
2      auto ints    = std::vector<int>{1, 2, 3};
3      auto doubles = std::vector<double>();
4
5      doubles.insert(
6              doubles.end(),
7              ints.begin(), ints.end()
8      );
9  }
```

Oznacza to, że metoda `.insert()` dla kontenerów z przestrzeni nazw `masters` powinna akceptować dowolny zakres, którego elementy mogą być użyte do stworzenia elementów przechowywanych przez docelową kolekcję. Kilka przykładów zostało zaprezentowanych w Listing 80.

Listing 80: Przykłady użycia szablonu metody `masters::vector::erase()`.

```
1  struct ConvertibleToInt {
2      operator int() const { return 1; }
3  };
4
5  auto main() -> int {
6      auto ints          = masters::vector<int>();
7
8      auto oldInts       = std::vector<int>();
9      auto doubles       = masters::vector<double>();
```

```
10      auto convertibles = masters::vector<ConvertibleToInt>();
11
12      ints.insert(ints.end(), oldInts);
13      ints.insert(ints.end(), doubles);
14      ints.insert(ints.end(), convertibles);
15  }
```

Implementacja tego szablonu również nie jest skomplikowana, ale wymaga już zdefiniowania pomocniczego narzędzia w postaci Konceptu.

Listing 81: Implementacja uzakresowionej wersji metody `.insert()`.

```
1  template <convertible_to_range_of<T> SourceRange>
2  auto insert(const_iterator pos, SourceRange&& rng) -> auto {
3      return insert(
4              pos,
5              std::ranges::begin(rng), std::ranges::end(rng)
6      );
7  }
```

Podobnie jak w przypadku strategii implementacji zaprezentowanej w Listing 77, Listing 81 prezentuje implementację polegającą na delegacji. Odziedziczona metoda `.insert()` również jest szablonem. Jedyna kluczowa rzecz to zdefiniować wymóg dotyczący zakresu, który chce się dodać do danego kontenera. Przedstawiony Koncept `convertible_to_range_of` ma za zadanie być spełniany tylko przez typy, które są zakresami, a elementy tych zakresów mogą zostać użyte do stworzenia elementów typu `T`, gdzie `T` to typ elementu kontenera, w kontekście którego zdefiniowany jest omawiany szablon `.insert()`.

W skrócie, można interpretować te implementację w następujący sposób: `SourceRange` musi być zakresem, którego elementy można skopiować / przekonwertować i wstawić do kontenera na dane miejsce.

Implementacja Konceptu `convertible_to_range_of` została przedstawiona w Listing 82.

Listing 82: Implementacja Konceptu `convertible_to_range_of`.

```
1  namespace masters {
2      template <typename Range, typename DestinationElementType>
3      concept convertible_to_range_of =
4      std::ranges::input_range<Range> and
5      std::convertible_to<
6              std::ranges::range_value_t<Range>, DestinationElementType
7      >;
8  }
```

Większość metod wylistowanych w rozdziale 6.2 może być zaimplementowana przy pomocy zaprezentowanej logiki i delegacji z tego podrozdziału. Niestety nie wszystkie. Konstruktor, nawet z nazwy, jest metodą specjalną.

Listing 80 prezentuje przykłady stworzenia pustych kolekcji i dodawania różnych elementów do nich. Jest to bardzo przydatna operacja, ale projektując narzędzia omawiane w tym rozdziale warto mieć na uwadze wygodę użytkowania. Typ `std::initializer_list` został wprowadzony do C++11 aby móc stworzyć wypełniony początkowymi wartościami kontener (za pomocą składni `{…, …, …}`) zamiast wymuszać tworzenie pustej kolekcji, a następnie dodawanie do niej elementów (np. za pomocą `.push_back()`).

Podobnie jak konstrukcja na podstawie listy elementów w klamrach, konstrukcja na podstawie innego zakresu powinna być udostępniona w celu zwiększenia wygody użytkowania tworzonego narzędzia. Przykład został zaprezentowany w Listing 83.

Listing 83: Przykład użycia konstruktorów konwertujących całe zakresy.

```
1  struct ConvertibleToInt {
2      operator int() const { return 1; }
3  };
4
5  auto main() -> int {
6      auto oldInts      = std::vector<int>();
7      auto doubles      = masters::vector<double>();
8      auto convertibles = masters::vector<ConvertibleToInt>();
9
10     auto v1 = masters::vector<int>(oldInts);
11     auto v2 = masters::vector<int>(doubles);
12     auto v3 = masters::vector<int>(convertibles);
13 }
```

Taki koncept abstrakcji nie jest niczym nowym. Języki programowania typu Python czy Java wspierają te konstrukcje od dawna. Przykładowo, klasa `ArrayList<E>` z Javy posiada konstruktor akceptujący dowolną kolekcję elementów dziedziczących po `E` (`ArrayList(Collection<? extends E> c)`), a klasa `set` z języka Python akceptuje w argumencie dowolny obiekt, po którym można iterować.

Jak zatem odpowiednio zdefiniować taki konstruktor? Czy wystarczy wprowadzić typ szablonowy będący ograniczeniem takim jak wcześniej zaprezentowane, tj. `convertible_to_range_of<T> SourceRange`?

Niestety nie.

Dla szablonu `masters::vector`, pełną definicją konstruktora konwertującego zakres będzie implementacja zaprezentowana w Listing 84.

Listing 84: Implementacja konstruktora konwertującego zakres dla `masters::vector`.

```
1  template <convertible_to_range_of<T> SourceRange>
2  requires (not std::same_as<std::remove_cvref_t<SourceRange>, vector>)
3  explicit vector(SourceRange&& rng, Allocator const& allocator = Allocator())
           : base(std::ranges::begin(rng), std::ranges::end(rng), allocator) { }
4
```

Warto zaznaczyć, dlaczego potrzebna jest dodatkowa klauzula `requires`, która będzie wymagała, aby argument `SourceRange` nie był tego samego typu, co tworzony właśnie wektor. Powód dotyczy wydajności i poprawności, czyli efektywnie wszystkiego.

Kwestia poprawności dotyczy wyboru odpowiedniej metody specjalnej. W Listing 85 zaprezentowano prostą klasę i przykład tworzenia dwóch obiektów jej typu. Tworzenie tych obiektów obrazuje omawiany problem.

Listing 85: Przykład problemu ze stosowaniem szablonu konstruktora.

```
 1  struct Example {
 2      Example() = default;                // default
 3  
 4      Example(Example const&) = default;  // copy
 5  
 6      template <typename T>
 7      Example(T&&) { }                    // none
 8  };
 9  
10  auto main() -> int {
11      auto original = Example();
12      auto copy     = original;
13  }
```

Komentarzem *default* został oznaczony konstruktor domyślny. Ten konstruktor to metoda specjalna (ang. *special member function*). Komentarzem *copy* został oznaczony konstruktor kopiujący. To również metoda specjalna. Komentarzem *none* został oznaczony szablon konstruktora przyjmujący uniwersalną referencję (ang. *universal reference / forwarding reference*). Nie jest to metoda specjalna.

Zaskakującym jest fakt, że przy tworzeniu obiektu o nazwie `copy`, zostanie wywołany nie konstruktor kopiujący, a konstruktor oznaczony komentarzem *none*. To dlatego, że wydedukowane dopasowanie `Example&` lepiej pasuje do typu `copy` niż `Example const&`, które zostałoby użyte przy konstruktorze kopiującym.

Łatwo sobie wyobrazić sytuację, gdzie konstruktor kopiujący jest w stanie wydajniej skopiować obiekt niż wersja generyczna, stąd pojawia się również argument optymalności rozwiązania.

Problem ten został bardzo dobrze wytłumaczony w [46]. Przytoczony tam problem wskazujący na błędne użycie `std::is_same_v` (w tym przypadku zastąpionego Konceptem `std::same_as`) nie aplikuje się do tego przykładu, bo zakłada się, że nikt nie będzie publicznie dziedziczył z kolekcji z `masters`.

6.3.4 Zakresy z elementami będącymi r-wartościami

Analizując kod z Listing 83, logicznym byłoby założyć, że kod zaprezentowany w Listing 86 będzie legalny i będzie skutkował przeniesieniem elementów z jednego kontenera do drugiego.

Listing 86: Konstrukcja kontenera na podstawie przeniesienia innego zakresu.

```
1  #include <vector>
2  #include <string>
3  #include <utility>  // std::move
4
5  #include <s17137/sequential/vector.hpp>
6
7  auto main() -> int {
8      auto strings = std::vector<std::string>{
9          "first", "second", "third"
10     };
11
12     auto moved = masters::vector<std::string>(std::move(strings));
13 }
```

Przeniesienie wektora `strings` powinno – zgodnie z intuicją – poskutkować przeniesieniem wewnętrznej reprezentacji tego kontenera do nowotworzonego `moved`. Tak się jednak nie dzieje.

Następnym założeniem mogłaby być chęć przeniesienia samych elementów ze `strings` do `moved`. Oryginalny wektor pozostałby niezmieniony w kontekście swojej reprezentacji. Tylko jego elementy byłyby przeniesione. Niestety, w tym przypadku również efekt jest inny od spodziewanego.

Biblioteka standardowa nie wspiera mechanizmów, które by pozwoliły na automatyczną identyfikację, kiedy iterując przez dany kontener powinno się przenosić jego elementy. Nie istnieją przeciążenia metod `.begin()`, którą są kwalifikowane na r-wartości[31].

Rozwiązaniem tego problemu do tej pory była funkcja-fabryka `std::make_move_iterator()`, która zwracała opakowanie na iterator, którego dereferencja skutkowała przeniesieniem elementu.

Jak natomiast wygląda sytuacja w przypadku reprezentowania zakresów w postaci pojedynczego obiektu zamiast pary iteratorów? Wspomniana fabryka adaptuje iterator, a nie cały zakres.

Warto zwrócić uwagę na celowo użyty termin *adaptuje* w poprzednim zdaniu. W rozdziale **4** zostały omówione adaptory zakresów, których natura pozwala dokładnie na mechanizm, który jest tutaj pożądany. Niestety adaptor, który udostępnia omawianą funkcjonalność, nie jest dostępny w C++20. Został dodany w C++23 pod nazwą `std::views::as_rvalue` [47]. To jego należy użyć (adaptując argument konstruktora i kompletnie rezygnując z `std::move()`), aby uzyskać pożądany efekt.

6.3.5 Problem z operatorami porównującymi

Kolekcje w C++ domyślnie wspierają wszystkie operatory porównania, tj. `==`, `!=`, `<`, `<=`, `>` oraz `>=`. Respektują one strategię porównywania leksykograficznego, czyli element po elemencie. Dzięki temu, w kodzie zaprezentowanym w Listing 87 zmienna `result` przechowuje wartość `true`.

[31] Tak samo jak metoda może być opatrzona kwalifikatorem `const`, może również być opatrzona kwalifikatorem kategorii wartości. Dzięki temu można dodać przeciążenie, które będzie wykonywane tylko na r-wartościach.

Listing 87: Przykład użycia operatorów porównujących dla typu `std::vector`.

```
1 auto main() -> int {
2     auto vec1 = std::vector<int>{1, 2};
3     auto vec2 = std::vector<int>{1, 2, 3};
4     auto vec3 = std::vector<int>{2, 3};
5
6     auto result = vec1 < vec2 and vec2 < vec3;
7 }
```

Problem w tym, że gdy w Listing 87 zastąpi się `std::vector` szablonem `masters::vector`, kod ten się nie skompiluje. Operatory porównania dla kolekcji do standardu C++20 były zdefiniowane jako wolne funkcje, które porównywały typy z biblioteki standardowej, a nie typy zdefiniowane w przestrzeni nazw `masters`. Każdy standardowy kontener posiadał przytoczony wcześniej zestaw operatorów, dzięki czemu można było łatwo je porównywać.

Aby uzyskać pożądane wsparcie tych operatorów, należy je samemu zaimplementować. Aby zachować spójność z konwencją biblioteki standardowej, która zmieniła się w C++20, powinien zostać zdefiniowany pojedynczy operator w postaci wolnej funkcji – `operator<=>` [48]. To znaczne ułatwienie w porównaniu do potrzeby zaimplementowania wszystkich 6 operatorów porównujących.

6.3.5.1 Podejścia różnych języków do rezultatu porównywania

Java, C++, Python i wiele innych języków wspierają podstawowe operatory porównywania. W każdym z nich porównanie – przykładowo – dwóch liczb całkowitych za pomocą operatora `<` zwróci zmienną logiczną o wartości prawdy lub fałszu. Inaczej natomiast wygląda sytuacja w przypadku porównywania typów innych niż prymitywne.

W języku Java szeroko stosowane są interfejsy `Comparable` oraz `Comparator`, które mają na celu zunifikowanie abstrakcji porównywania obiektów. `Comparable` jest stosowany w przypadkach, gdy programista chce, aby dany typ wspierał jakiś domyślny sposób porządkowania elementów. `Comparator` jest zwykle używany w przypadkach, gdzie logika modelująca dany porządek ma być unikalna dla lokalnego wykorzystania.

Przykładowo, klasa `String` w Javie implementuje interfejs `Comparable`, w ramach którego definiuje porządek leksykograficzny, a z uwagi na to, że elementy obiektów typu String to znaki, efektywnie reprezentuje to kolejność alfabetyczną. Jest to domyślny sposób na porządkowanie `String`ów.

Czasami jednak programista potrzebuje uporządkować je (np. algorytmem sortującym) według jakiejś innej logiki, np. od najkrótszego do najdłuższego. Nie da się wpłynąć na to, jak `Comparable` został zaimplementowany w `String`, ale algorytmy sortujące zawsze akceptują dodatkowy argument w postaci obiektu typu `Comparator`, który pozwala na zdefiniowanie lokalnej logiki, według której dane obiekty będą uporządkowane.

Kontekst tego przykładu został już wcześniej zobrazowany – w rozdziale **2** w ramach omówienia Listing 3.

Warto jednak dokładnie przyjrzeć się modelowi abstrakcji porównywania w Javie. Opiera się ona na zwracaniu wartości całkowitoliczbowej – `inta` – jako reprezentacji relacji. Gdy porównane zostaną dwa obiekty (czy to za pomocą logiki z `Comparable`, czy z `Comparator`), zwrócony `int` przyjmuje jedną z trzech grup wartości:

- `0`, jeżeli obiekty są uznawane za „równe”;
- liczbę mniejszą niż `0`, jeżeli lewy argument jest uznawany za „mniejszy”;
- liczbę większą niż `0`, jeżeli lewy argument jest uznawany za „większy” (i powinny zostać zamienione miejscami).

Sprawia to, że kompletność informacji o relacji może być zawarta w pojedynczej wartości.

W języku C++ jest inaczej. Komparatory wykorzystywane przez bibliotekę standardową zakładają, że narzędzia te zwracają wartość logiczną `bool`. W przypadku, gdy lewy argument jest „mniejszy”, zwracana jest prawda. W odwrotnym przypadku fałsz. Warto zwrócić uwagę na negację tej relacji – negacją *mniejszości* nie jest *większość*, a relacja *mniejszy lub równy*. Oznacza to, że gdy komparator zwraca `false`, to algorytm wie jedynie, że drugi argument nie jest mniejszy od pierwszego. Oznacza to, że mogą być równe lub prawy jest mniejszy.

Nie jest to jednak problemem, choć niektórzy programiści potrafią popełnić błąd w postaci zdefiniowania komparatora na podstawie logiki relacji mniejszy-równy, co formalnie skutkuje zachowaniem niezidentyfikowanym (UB).

Podsumowując, aby dany typ zdefiniowany przez użytkownika wspierał poprawną i pełną relację porządkową, należy dostarczyć implementacje wspomnianych na początku rozdziału sześciu operatorów. W momencie, gdy programista chce skorzystać z unikalnej relacji, np. w algorytmie `std::ranges::sort()` lub `std::ranges::max()`, musi dostarczyć komparator, który modeluje relację mniejszości.

W tej sekcji pracy idea oscyluje wokół tego pierwszego problemu, czyli dostarczenia domyślnych operatorów porządkowych. Jak już zostało to wspomniane, dzięki nowemu, dodanemu w C++20 operatorowi `<=>`, nie trzeba tworzyć aż sześciu implementacji. Wystarczy jedna.

6.3.5.2 Implementacja operatorów porównujących dla kontenerów z `masters`

Aby móc do woli porównywać obiekty typu `masters::vector` między sobą, z założenia wystarczyłoby zdefiniować szablon przedstawiony w Listing 88.

Listing 88: Nagłówek szablonu operatora `<=>` dla `masters::vector`.

```
1  namespace masters {
2      template <typename T, typename Allocator>
3      auto operator<=>(
4              vector<T, Allocator> const& lhs,
```

```
5               vector<T, Allocator> const& rhs
6       ) -> auto { … }
7   }
```

Obiekt zwracany przez `<=>`, zgodnie z jego specyfikacją, pozwoli na weryfikację relacji zachodzącej między `lhs` i `rhs` (lewy i prawy operand). Nie wchodząc w szczegóły, dzięki niemu będzie można wywołać dowolny operator relacji z operandami `lhs` i `rhs`. Ten operator zostanie przekształcony w czasie kompilacji na odpowiednie użycie obiektu zwracanego przez `<=>`.

Implementacja „od zera” nie byłaby szczególnie trudna, ale warto wpierw odpowiedzieć na następujące pytanie: czy na pewno trzeba implementować ten operator od zera?

Nie trzeba. Można delegować jego implementację do istniejącego już operatora `<=>`. Takiego, który został zdefiniowany dla `std::vector`.

Problem w tym, że dostarczony przez standardową bibliotekę `operator<=>` przyjmujący dwa obiekty typu `std::vector<T>` nie może zostać użyty do porównania dwóch obiektów typu `masters::vector<T>`. To dlatego, że o ile `masters::vector<T>` dziedziczy z `std::vector<T>`, to nie dziedziczy on z niego publicznie.

Nie dyskwalifikuje to jednak strategii delegującej implementację omawianego operatora do tej udostępnianej przez bibliotekę standardową. Wystarczy w pewnym sensie oszukać system typów w C++ i wskazać, że jako autorzy tego rozwiązania, jesteśmy całkowicie pewni, że legalnym jest w tym konkretnym przypadku złamać hermetyzację i wymusić akceptację obiektu typu `masters::vector<T>` tam, gdzie kod oczekuje `std::vector<T>`. Jest to jeden z niewielu przypadków, gdzie `reinterpret_cast<>()` znajduje zastosowanie. Zostało to ukazane w Listing 89.

Listing 89: Implamentacja szablonu operatora `<=>` dla `masters::vector`.

```
1   namespace masters {
2       template <typename T, typename Allocator>
3       auto operator<=>(
4               vector<T, Allocator> const& lhs,
5               vector<T, Allocator> const& rhs
6       ) -> auto {
7           using base = std::vector<T, Allocator>;
8           auto const& baseLhs
9                   = reinterpret_cast<base const&>(lhs);
10          auto const& baseRhs
11                  = reinterpret_cast<base const&>(rhs);
12          return baseLhs <=> baseRhs;
13      }
14  }
```

Mówi się czasem, że „programista C++ ma zawsze rację”. Wykorzystanie bardziej wyspecjalizowanych narzędzi jawnego rzutowania i konwersji (np. `reinterpret_cast<>()` czy `const_cast<>()`) jest bardzo ryzykowną i zwykle odradzaną praktyką [49], ale w takich przypadkach jak ten oczywistym się staje, że owa praktyka jest czasem najlepszym narzędziem do osiągnięcia obranego celu.

Taki mechanizm pozwala na wykorzystanie `<`, `<=`, `>` i `>=` dla `masters::vector`. Niektórzy mogą tutaj zapytać, dlaczego `==` i `!=` zostały pominięte w tej liście. `operator<=>` pozwala na ich generowanie, ale w przypadku kolekcji generowałby on nieco nieoptymalny kod (przy równości i nierówności warto wpierw sprawdzić liczbę elementów z uwagi na to, że gdy ona się różni między dwoma obiektami, to nie mogą być równe). Dlatego `std::vector` (i inne kolekcje) używają `<=>` do generowania operatorów poza `==` i `!=`. Do generowania ostatnich dwóch wystarczy zdefiniować pojedynczy `==`. Ten ostatni również pozwala na generowanie i wykorzystywanie `!=`.

`operator<=>` wprowadza jeszcze kilka dodatkowych udogodnień, przykładowo w postaci symetrycznego generowania operatorów (tj. dla porównywania dwóch obiektów o różnych typach już nie trzeba duplikować implementacji). Nie jest to jednak kluczowe dla pracy, dlatego te szczegóły zostały pominięte.

Na koniec warto jeszcze dodać, że autor pracy rozważał zaimplementowanie `<=>` oraz `==` dla dowolnej kombinacji szablonów kolekcji z przestrzeni nazw `masters`. Dzięki temu możliwym byłoby stworzyć dwa osobne kontenery (np. `masters::set<int>` oraz `masters::vector<double>`) i, o ile ich elementy byłyby porównywalne ze sobą, porównywać je leksykograficznie.

Pomysł ten został jednak porzucony z dwóch powodów:

1. Narzędzie do porównywania arbitralnych zakresów już istnieje. Nazywa się `std::lexicographical_compare()` i `std::lexicographical_compare_three _way()`. Pozwala na sprecyzowanie dowolnych zakresów (jako pary iteratorów, co można zauważyć obserwując brak podprzestrzeni nazw `ranges`) i porównanie ich leksykograficznie.
2. Zestaw operatorów do porównywania każdej kombinacji dwóch kolekcji byłby w pewien sposób sprzeczny z ideą modelowania relacji równości traktującej o identyczności obiektów. O ile porównanie obiektów o różnych typach miewa sens (patrz – abstrakcja strażnika (ang. *sentinel*) z Zakresów [21]), to jest to poniekąd relikt przeszłości. Dług techniczny, który został z C++ z powodu pierwotnie podjętych decyzji, które nie przewidziały późniejszych potrzeb.

6.3.6 Implementacja wskazówek dedukcji argumentów szablonowych dla klas (CTAD)

Wskazówki dedukcji argumentów szablonowych klas, w (angielskim) skrócie CTAD, są przytoczonym już wcześniej w pracy (koniec rozdziału **6.2**) narzędziem pomagającym w pominięciu precyzowania argumentów szablonowych dla klas. Dzięki temu, zamiast precyzować argument szablonowy `int` przy `std::vector` tworząc od razu kolekcję kilku liczb całkowitych (tak jak to przedstawiono w Listing 90), można zastosować składnię zaprezentowaną w Listing 91.

Listing 90: „Klasyczna" deklaracja obiektu typu `std::vector<int>`.

```
1 auto main() -> int {
2     auto vec = std::vector<int>{1, 2, 3};
3 }
```

Listing 91: Deklaracja obiektu typu `std::vector<int>` z pominięciem argumentu szablonu precyzującego typ elementu.

```
1 auto main() -> int {
2     auto vec = std::vector{1, 2, 3};
3 }
```

Jak można przeczytać w [50], głównymi powodami wprowadzenia CTAD było uproszczenie składni w nietrywialnych miejscach, takich jak:

- Typy `std::lock_guard`, którym często trzeba było sprecyzować oczywisty w kontekście ich użycia konkretny wariant mutexów.
- Typy, których konstruktor przyjmował lambdy.
- Niepotrzebne zaśmiecanie biblioteki standardowej przeróżnymi funkcjami-fabrykami typu `std::make_pair()` czy `std::make_tuple()`.

Ten ostatni podpunkt dosyć dokładnie obrazuje ogromną, pierwotną różnicę między szablonami funkcji i szablonami klas. Parametry szablonowe w szablonach funkcji mogły być dedukowane na podstawie faktycznych argumentów przesłanych podczas ich wywoływania, gdzie w przypadku szablonów klas nie było w ogóle takiej możliwości. To właśnie geneza powstania niektórych wspomnianych funkcji-fabryk.

Dzięki CTAD wiele z tych problemów zostało zażegnane. Warto w takim razie wzbogacić szablony z `masters` o możliwość tworzenia ich obiektów bez precyzowania konkretnego typu elementów. Powinien on mieć możliwość bycia wydedukowanym na podstawie typu elementu źródłowego zakresu, na podstawie którego tworzona jest kolekcja.

W tym miejscu warto zapytać, czy przypadkiem ta funkcjonalność nie jest automatycznie dostarczana z uwagi na dziedziczenie po standardowych kontenerach? Czy nie wystarczy upublicznić odpowiednich konstruktorów lub wskazówek tak, jak zostało to omówione w rozdziale **6.3.2**? Niestety nie. Problem ten został rozwiązany dzięki adaptacji wniosku [51] bazującego na [52], który jednak dotyczy wyłącznie standardów od C++23 wzwyż. W związku z tym, że niniejsza praca koncentruje się na C++20, implementacja została wzbogacona o odpowiednie wskazówki, o których była mowa w tym rozdziale. Zostały one przedstawione w Listing 92.

Listing 92: Implementacja wskazówek dedukcji argumentów szablonowych dla klas (CTAD).

```
1  namespace masters {
2      template <typename T, typename Allocator = std::allocator<T>>
3      vector(
4              typename std::vector<T, Allocator>::size_type,
5              T const&,
6              Allocator const& = Allocator()
7      ) -> vector<T, Allocator>;
8
9      template <
10             typename InputIt,
11             typename Allocator = std::allocator<
12                     typename std::iterator_traits<InputIt>::value_type
13             >
14     >
15     vector(
16             InputIt, InputIt, Allocator = Allocator()
17     ) -> vector<
18             typename std::iterator_traits<InputIt>::value_type, Allocator
19     >;
20
21     template <
22             std::ranges::range SourceRange,
23             typename Allocator = std::allocator<
24                     std::ranges::range_value_t<SourceRange>
25             >
26     >
27     vector(
28             SourceRange&&, Allocator = Allocator()
29     ) -> vector<
30             std::ranges::range_value_t<SourceRange>, Allocator
31     >;
32 }
33
```

Logika implementacji jest bardzo prosta. Dla każdego konstruktora należy powtórzyć jest deklarację szablonową i sprecyzować (po tokenie `->`) typ, który powinien być wtedy wydedukowany.

Większość z zaprezentowanych w Listing 92 wskazówek jest powtórzona z szablonu `std::vector`. Wyjątkiem jest ostatnia z nich, dodatkowo wyeksponowana w Listing 93.

Listing 93: Unikalny CTAD dla `masters::vector`.

```
1  template <
2          std::ranges::range SourceRange,
3          typename Allocator = std::allocator<
4                  std::ranges::range_value_t<SourceRange>
5          >
6  >
7  vector(
8          SourceRange&&, Allocator = Allocator()
9  ) -> vector<
10
```

```
11          std::ranges::range_value_t<SourceRange>, Allocator
    >;
```

Warto zwrócić uwagę na ścisłe powiązanie narzędzi z C++20. Dzięki omówionemu już `std::ranges::range_value_t` można bardzo łatwo wydedukować, jaki rodzaj kolekcji powinien zostać stworzony. Listing 83 prezentował konstrukcję kolekcji na podstawie abstrakcji zakresu wyrażonej w postaci pojedynczego obiektu. Dzięki dodanemu CTAD można w nim pominąć argumenty szablonowe tak, jak zostało to zaprezentowane w Listing 94.

Listing 94: Przykład użycia konstruktorów konwertujących całe zakresy z pominięciem argumentu szablonowego precyzującego typ elementu kolekcji.

```
1  auto main() -> int {
2      auto oldInts      = std::vector<int>();
3      auto doubles      = masters::vector<double>();
4      auto convertibles = masters::vector<ConvertibleToInt>();
5
6      auto v1 = masters::vector(oldInts);
7      auto v2 = masters::vector(doubles);
8      auto v3 = masters::vector(convertibles);
9  }
```

Taki mechanizm CTAD jest niezwykle przydatny głównie w generycznym kontekście. Gdy otrzyma się arbitralny zakres i chce się z niego skonstruować jeden z kontenerów z przestrzeni nazw `masters`, nie trzeba wykorzystywać wyszukanych technik metaprogramistycznych (nawet jeżeli ograniczają się one do pojedynczego wykorzystania `std::ranges::range_value_t` – wtedy zwykle i tak trzeba sprecyzować dłuższe argumenty dla tego narzędzia). Wystarczy podać ten zakres do konstruktorów danego typu.

Warto na koniec zaznaczyć, że w Listing 92 nie znajdziemy CTAD dla konstruktora przyjmującego `std::initializer_list`. To dlatego, że jest on domyślnie generowany. Wystarczy zdefiniować konstruktor przyjmujący taką listę. Przykład został zaprezentowany w poprawnie kompilującym się Listing 95.

Listing 95: Przykład automatycznie wygenerowanego CTAD dla konstruktora wykorzystującego `std::initializer_list`.

```
1  #include <initializer_list>
2
3  template <typename T>
4  struct Example {
5      Example(std::initializer_list<T>) { }
6  };
7
8  auto main() -> int {
9      auto example = Example{1, 2, 3};
10 }
```

6.3.7 Pozostałe kontenery z masters

W rozdziale **6.2** poza `std::vector` zostało zidentyfikowane wiele innych szablonów kontenerów, które zasługują na otrzymanie swoich usprawnionych wersji w przestrzeni nazw `masters`. Jednakże większość z nich byłaby implementowana w bardzo podobny sposób, tj. korzystając z prywatnego dziedziczenia, upubliczniając poszczególne, odziedziczone elementy oraz rozszerzając funkcjonalności w postaci nowych konstruktorów i nowych przeciążeń metod.

Dlatego zamiast prezentować powtarzający się tendencyjnie kod, warto zidentyfikować resztę elementów jako implementowalne w sposób trywialny, zakładając dobrą znajomość mechanizmów wprowadzonych w poprzednich rozdziałach. Dodatkowo warto omówić niektóre metody, które nie doczekały się procesu uzakresowienia w kontekście niniejszej pracy.

Na początek należy przyjrzeć się kontenerom asocjacyjnym z przedrostkiem `multi`. Są to: `std::multiset`, `std::unordered_multiset`, `std::multimap` oraz `std::unordered_mult imap`. Ich specjalną cechą jest to, że pozwalają na duplikaty elementów. W przypadku map są to oczywiście duplikaty kluczy. Do swoich „klasycznych" odpowiedników dodają metodę `.equal_range()`, która zwraca zakres elementów równych szukanemu elementowi / kluczowi.

Aby zachować spójność z ideą tworzenia rozszerzeń z `masters`, oczywistym się wydaje, że należałoby dodać nowe przeciążenie tej metody. Takie, które będzie zwracało `std::ranges::subrange` zamiast „przestarzałej" reprezentacji zakresu będącego parą (`std::pair`) dwóch iteratorów.

Niestety, nie jest to możliwe w języku C++. To dlatego, że język ten nie pozwala na przeciążanie funkcji lub metod na typie zwracanym. Jedynym sposobem na dostarczenie omawianej, dodatkowej funkcjonalności byłoby wprowadzenie nowej metody – z inną nazwą.

Jest to przykład przewagi alternatywnego rozwiązania przedstawionego w rozdziale **6.3** nad obranym w tej pracy. W przypadku modyfikowania biblioteki standardowej lub tworzenia własnych kontenerów można by kompletnie zastąpić oryginalne wersje tych metod nowymi.

Ich implementacja nie byłaby jednak jakkolwiek kłopotliwa. Ponownie wykorzystany byłby tutaj model delegacji, gdzie otrzymana para z „oryginalnej" wersji zostałaby użyta do stworzenia obiektu typu `std::ranges::subrange`. W związku z tym, nie została ona zaprezentowana w niniejszej pracy.

7 Testowanie stworzonego narzędzia

Istotnym elementem wytwarzania oprogramowania jest jego testowanie. Robert C. Martin w Clean Code [9] sugeruje nawet adaptację podejścia TDD, czyli Test-Driven Development (wytwarzanie oprogramowania w oparciu o testy). Podejście to zakłada, że zamiast wytwarzać oprogramowanie a potem je testować, należy najpierw stworzyć odpowiedni zestaw testów, a dopiero później pisać w oparciu o nie kod.

Dywagacje dotyczące słuszności opisanego podejścia nie są jednak tematem niniejszej pracy. Warto jednak rozważyć przetestowanie nowostworzonych narzędzi za pomocą jakiegoś ustalonego zbioru testów, aby wykazać poprawność omawianego efektu.

W roku 2018 firma Microsoft wykupiła platformę GitHub – jednego z najpopularniejszych serwisów hostujących repozytoria programistyczne. Jest to istotny fakt z uwagi na to, że dzięki temu Microsoft zdecydował się tam upublicznić kod swojej biblioteki standardowej, która jest częścią zestawu MSVC (Microsoft Visual C++) – kompilatora do języków C, C++, C++/CLI oraz C++/CX.

Został tam również udostępniony ich pełen zestaw testów [53]. Wyodrębnienie tych, które dotyczą kolekcji, z których szablony z `masters` dziedziczą, jest dosyć prostym procesem. Tak przeprowadzone testy (z wykorzystaniem skryptów Pythonowych oraz systemu metabudowania CMake) zweryfikowały pełną poprawność implementacji stworzonej w ramach niniejszej pracy.

Jest to istotny przykład zachowania pełnej poprawności zachowań dzięki jawnemu delegowaniu implementacji w postaci dziedziczenia z selektywnym, kontrolowanym sterowaniem interfejsów.

8 Alternatywne rozwiązania

W podobnym czasie do tego, w którym niniejsza praca zaczynała być tworzona, pojawił się wniosek dotyczący konwersji zakresów do kontenerów [54]. Traktuje on o dodaniu dwóch nowych narzędzi:

1. `std::ranges::to`, które pozwoli na materializację sprecyzowanego kontenera na podstawie arbitralnego zakresu;
2. nowych konstruktorów do kontenerów ze standardowej biblioteki, które wykorzystując mechanizm tagowania (ang. *tag dispatch*) będą implementowały logikę zamiany zakresu na kontener.

Efektywnie, pierwsze narzędzie będzie delegowało część pracy do drugiego.

Dodatkowo, wniosek mówi o dodaniu nowych metod `.insert()` oraz `.assign()`, które zachowywały by się tak, jak te sprecyzowane dla kolekcji z przestrzeni nazw `masters`. Różnica polega jednak na tym, że metody te nie byłyby przeciążeniami, bo otrzymałyby inne nazwy, takie jak `.insert_range()`, `.assign_range()` czy `.push_front_range()`.

W jakim stopniu zatem implementacja wykonana na rzecz niniejszej pracy różni się od tej, która powinna zostać wykonana w ramach wprowadzenia wspomnianego wniosku w życie?

Przede wszystkim należy zwrócić uwagę, że wniosek nie zakłada żadnej konkretnej implementacji. Można zatem uznać, że narzędzia z `masters` są w pewien sposób jej stanem sztuki. Niemniej jednak należy przyjrzeć się szczegółom implementacyjnym w bibliotekach standardowych implementowanych przez wiodących dystrybutorów.

8.1 Implementacja MSVC

Implementacja Microsoftu jest bardzo podobna do implementacji zaprezentowanej w niniejszej pracy. Korzysta ona z bliźniaczego do `convertible_to_range_of` Konceptu o nazwie `_Container_compatible_range`. Efektywnie, te Koncepty są identyczne.

Implementacje są do siebie bardzo zbliżone pod wieloma względami. Główna różnica polega na tym, że wersja stworzona przez Microsoft nie deleguje pracy do istniejących już metod. Po analizie kodu można stwierdzić, że powodem jest potencjalnie nieoptymalna implementacja metody `.insert()`. Nie wykorzystuje ona mechanizmu pojedynczej alokacji na dodawane elementy w przypadku, co zostało naprawione w nowododanej metodzie.

8.2 Implementacje pozostałych wiodących dystrybutorów kompilatorów i bibliotek standardowych

W momencie pisania niniejszej pracy żadna inna implementacja biblioteki standardowej nie wspiera zmian wprowadzonych przez [54].

9 Podsumowanie i wnioski

Programowanie generyczne jest nieodzownym elementem wytwarzania oprogramowania. Ukazuje ono tendencję, która towarzyszy programistom w większości projektów – aby stworzyć narzędzie, które będzie służyło długo i efektywnie, należy włożyć trochę więcej wysiłku i pracy w proces jego projektowania. Narzędzia generyczne w pewien sposób odzwierciedlają naturę narzędzi biznesowych, mimo tego, że są to dwa odległe spektra tworzenia aplikacji. Wzorce projektowe pomagają stworzyć łatwe w rozszerzaniu moduły, a dobrze zaprojektowane narzędzia generyczne pozwalają na proste, intuicyjne, poprawne i wydajne wykonywanie powtarzalnych czynności.

Wiele współczesnych języków programowania rozwiązuje problemy wytwarzania generycznego kodu w unikalny dla siebie sposób. Niemniej jednak każdy z nich w jakiś sposób albo takie programowanie wspiera, albo takie wsparcie emuluje. Wsparcie, jakie dany język programowania oferuje do programowania generycznego waha się między integralną częścią języka, która jest ledwie pośrednio związaną z takim programowaniem, a obszernym zestawem dedykowanych narzędzi.

Jednym z najważniejszych aspektów programowania generycznego jest elastyczność w kontekście używalności danych narzędzi. Generyczność to nie tylko możliwość dopasowania się do każdego rodzaju (lub raczej – większości rodzajów) danych, ale też umożliwienie programiście sterowanie elementarnymi mechanizmami tych narzędzi.

Kolejnym istotnym aspektem jest zgłaszanie błędów użycia narzędzi generycznych. Gdy narzędzie jest użyte niepoprawnie, użytkownik powinien otrzymać jasny i klarowny opis błędu, który – w idealnym świecie – dodatkowo sugerowałby jak dany błąd naprawić.

Język C++ jest dosyć osobliwym przypadkiem. Wyróżnia się ogromną liczbą narzędzi wspierających programowanie generyczne – od narzędzi językowych takich jak szablony, Koncepty, specjalne notacje funkcji aż po narzędzia biblioteki standardowej, takie jak `<type_traits>`, `enable_if`, `<tuple>` i inne.

Wynika to z idei, która kształtuje ten język. Narzędzia generyczne powinny być maksymalnie wydajne. Czasem jednak sprawia to, że są one bardzo trudne w użyciu, a jeszcze trudniejsze w implementacji. Ich poziom skomplikowania przekłada się bezpośrednio na dwa, kluczowe dla programistów czynniki:

- trudne do interpretacji błędy kompilacji;
- ubogość biblioteki standardowej, która przecież powinna być bogata w narzędzia ułatwiające pracę programisty.

Niemniej jednak nie sposób zauważyć, że wraz z nowymi standardami języka, a zwłaszcza w epokach nowoczesnego i kompletnego C++, język ten jest zdecydowanie prostszy i przyjemniejszy w użyciu. Nowe abstrakcje niosą za sobą również nowe terminologie, dzięki którym łatwiej można precyzować i kształtować dalszą ewolucję tego narzędzia. Dobrym tego przykładem jest stworzona w ramach pracy implementacja rozszerzająca bibliotekę standardową języka C++. Obszerne używanie nowowprowadzonych narzędzi jasno wskazuje na ich użyteczność, a omawiane alternatywy (również historyczne) prezentują dobry obraz ewolucji języka. Prezentuje ona zarówno potrzebę jak i przepaść funkcjonalną

między oryginalnymi założeniami a narzędziami, które powstały w wyniku weryfikacji potrzeb biznesowych.

Prace cytowane

[1] „StackOverflow Developer Survey," 2023. Dostęp: Lipiec 2023. https://survey.stackoverflow.co/2023/#most-popular-technologies-language

[2] S. Meyers, „Designing Interfaces," w *DevTraining*. https://youtu.be/TdajK_SXwoc Dostęp: Lipiec 2023.

[3] R. W. Sebesta, Concepts of Programming Languages. Pearson 2016. ISBN 13: 978-1-292-10055-5.

[4] Andy Hunt, Dave Thomas, The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional 1999. ISBN 13: 978-0201616224.

[5] B. Venners, Interviewee, *Orthogonality and the DRY Principle, A Conversation with Andy Hunt and Dave Thomas, Part II.* [Wywiad]. 2003. https://www.artima.com/articles/orthogonality-and-the-dry-principle Dostęp: Lipiec 2023.

[6] S. Meyers, Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison Wesley 2005. ISBN 13: 978-0321334879.

[7] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Conallen, Kelli A. Houston, Object-Oriented Analysis and Design with Applications. Addison-Wesley Professional 2007. ISBN 13: 978-0201895513.

[8] J. Lakos, Large-Scale C++ Software Design. Addison-Wesley, 1996. ISBN 13: 978-0201633627.

[9] R. C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship. Pearson 2008. ISBN 13: 978-0132350884.

[10] B. Stroustrup, The Design and Evolution of C++, AT&T Bell Labs, 1994. ISBN: 0-201-54330-3

[11] H. Sutter, „GotW #94 Solution: AAA Style (Almost Always Auto)," 2013. https://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/ Dostęp: Lipiec 2023.

[12] H. Sutter, „Back to the Basics! Essentials of Modern C++ Style," w *CppCon*, 2014. https://youtu.be/xnqTKD8uD64 Dostęp: Lipiec 2023.

[13] J. Boccara, „"auto to stick" and Changing Your Style". https://www.fluentcpp.com/2018/09/28/auto-stick-changing-style/ Dostęp: Lipiec 2023.

[14] Jaakko Järvi, Bjarne Stroustrup, „Decltype and auto (revision 3), N1607=04-0047". https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2004/n1607.pdf Dostęp: Lipiec 2023.

[15] „Undefined Behavior - CppReference". https://en.cppreference.com/w/cpp/language/ub Dostęp: Lipiec 2023.

[16] S. Meyers, Effective Modern C++. O'Reilly Media, Incorporated 2015. ISBN 13: 978-1491903995.

[17] P. Zemek, „Pros and Cons of Alternative Function Syntax in C++”. https://blog.petrzemek.net/2017/01/17/pros-and-cons-of-alternative-function-syntax-in-cpp/ Dostęp: Lipiec 2023.

[18] A. Zhilin, „P2025R2: Guaranteed copy elision for return variables”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2025r2.html Dostęp: Lipiec 2023.

[19] A. Sutton, „Wording Paper, C++ extensions for Concepts”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0734r0.pdf Dostęp: Lipiec 2023.

[20] Casey Carter, Eric Niebler, „Standard Library Concepts”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0898r3.pdf Dostęp: Lipiec 2023.

[21] Eric Niebler, Casey Carter, Christopher Di Bella, „The One Ranges Proposal”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0896r4.pdf Dostęp: Lipiec 2023.

[22] R. Smith, „Merging Modules”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1103r3.pdf Dostęp: Lipiec 2023.

[23] Stephan T. Lavavej, Gabriel Dos Reis, Bjarne Stroustrup, Jonathan Wakely, „Standard Library Modules std and std.compat”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2465r3.pdf Dostęp: Lipiec 2023.

[24] G. Nishanov, „Working Draft, C++ Extensions for Coroutines”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4775.pdf Dostęp: Lipiec 2023.

[25] J. Boccara, „How to Use Tag Dispatching In Your Code Effectively”. https://www.fluentcpp.com/2018/04/27/tag-dispatching/ Dostęp: Lipiec 2023.

[26] R. Grimm, „Software Design with Traits and Tag Dispatching”. https://www.modernescpp.com/index.php/softwaredesign-with-traits-and-tag-dispatching Dostęp: Lipiec 2023.

[27] Ville Voutilainen, Daveed Vandevoorde, „constexpr if”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0128r1.html Dostęp: Lipiec 2023.

[28] J. Maurer, „P0292R1: constexpr if: A slightly different syntax”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0292r1.html Dostęp: Lipiec 2023.

[29] e. a. Alexander A. Stepanov, C++ Standard Template Library. Pearson 2000. ISBN 13: 978-0134376332.

[30] Alexander A. Stepanov, Daniel E. Rose, „Od matematyki do programowania uogólnionego”. HELION S.A 2015. ISBN: 978-83-283-1028-5.

[31] „Abbreviated function template”. https://en.cppreference.com/w/cpp/language/function_template#Abbreviated_function_template Dostęp: Lipiec 2023.

[32] „Kopia kodu źródłowego GCC wraz z implementacją biblioteki standardowej libstdc++”. https://github.com/gcc-mirror/gcc Dostęp: Lipiec 2023.

[33] „Constraints and concepts”. https://en.cppreference.com/w/cpp/language/constraints Dostęp: Lipiec 2023.

[34] T. Kamiński, „A proposal to add invoke function template (Revision 1)”. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4169.html Dostęp: Lipiec 2023.

[35] „StackOverflow: Using a comparator function to sort”. https://stackoverflow.com/questions/12749398/using-a-comparator-function-to-sort Dostęp: Lipiec 2023.

[36] X. Rigoulet, „How to Write Custom Sort Functions in Python”. https://learnpython.com/blog/python-custom-sort-function/ Dostęp: Lipiec 2023.

[37] F. Kwiatkowski, „Why doesn't std::ranges::upper_bound accept heterogenous comparing?,” 2021. https://stackoverflow.com/q/70363847/7151494 Dostęp: Lipiec 2023.

[38] Eric Niebler, Sean Parent, Andrew Sutton, „Ranges for the Standard Library, Revision 1,” 2014. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4128.html#algorithms-should-take-invokable-projections Dostęp: Lipiec 2023.

[39] „Java 8 Stream API”. https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html Dostęp: Lipiec 2023.

[40] „CppReference: Variable template”. https://en.cppreference.com/w/cpp/language/variable_template Dostęp: Lipiec 2023.

[41] „std::ranges::view, std::ranges::enable_view, std::ranges::view_base”. https://en.cppreference.com/w/cpp/ranges/view Dostęp: Lipiec 2023.

[42] „StackOverflow: How does "using std::swap" enable ADL?”. https://stackoverflow.com/q/28130671/7151494 Dostęp: Lipiec 2023.

[43] A. O'Dwyer, „What is the std::swap two-step?,” 2020. https://quuxplusone.github.io/blog/2020/07/11/the-std-swap-two-step/ Dostęp: Lipiec 2023.

[44] J. Turner, „C++ Weekly - Ep 231 - Multiple Destructors in C++20?! How and Why,” 2020. https://youtu.be/A3_xrqr5Kdw Dostęp: Lipiec 2023.

[45] R. Grimm, „Surprise Included: Inheritance and Member Functions of Class Templates,” 2021. https://www.modernescpp.com/index.php/surprise-included-inheritance-and-member-functions-of-class-templates Dostęp: Lipiec 2023.

[46] N. Josuttis, „CppCon 2017: Nicolai Josuttis “The Nightmare of Move Semantics for Trivial Classes”,” Bellevue, 2017. https://youtu.be/PNRju6_yn3o Dostęp: Lipiec 2023.

[47] B. Revzin, „views::as_rvalue,” 2022. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2446r2.html Dostęp: Lipiec 2023.

[48] Herb Sutter, Jens Maurer, Walter E. Brown, „Consistent comparison,” 2017. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0515r3.pdf Dostęp: Lipiec 2023.

[49] Bjarne Stroustrup, Herb Sutter, „Cpp Core Guidelines ES.50: Don't cast away const". https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#es50-dont-cast-away-const Dostęp: Lipiec 2023.

[50] Mike Spertus, Faisal Vali, Richard Smith, „Template argument deduction for class templates (Rev. 6)," 2016. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0091r3.html Dostęp: Lipiec 2023.

[51] T. Doumler, „Wording for class template argument deduction from inherited constructors," 2022. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2582r1.pdf Dostęp: Lipiec 2023.

[52] Mike Spertus, Timur Doumler, Richard Smith, „Filling holes in Class Template Argument Deduction," 2022. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1021r6.html Dostęp: Lipiec 2023.

[53] S. T. Lavavej, „MSVC STL Tests". https://github.com/microsoft/STL/tree/main/tests Dostęp: Lipiec 2023.

[54] Corentin Jabot, Eric Niebler, Casey Carter, „Conversions from ranges to containers," 2022. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p1206r7.pdf Dostęp: Lipiec 2023.

Spis rysunków

RYSUNEK 1: OKNO PODRĘCZNEJ DOKUMENTACJI WYWOŁANE NA RZECZ ZMIENNEJ VALUE W ŚRODOWISKU CLION. .. 31
RYSUNEK 2: OKNO PODRĘCZNEJ DOKUMENTACJI WYWOŁANE NA RZECZ ZMIENNEJ VALUE W ŚRODOWISKU VISUAL STUDIO. 32
RYSUNEK 3: PRZYKŁADOWA HIERARCHIA PARADYGMATÓW JĘZYKÓW PROGRAMOWANIA. 65
RYSUNEK 4: LISTA DOSTĘPNYCH FUNKCJONALNOŚCI PODPOWIADANYCH PRZEZ ŚRODOWISKO JETBRAINS INTELLIJ IDEA DLA OBIEKTU TYPU GENERYCZNEGO W JAVIE. 76
RYSUNEK 5: LISTA DOSTĘPNYCH FUNKCJONALNOŚCI PODPOWIADANYCH PRZEZ ŚRODOWISKO JETBRAINS CLION DLA OBIEKTU TYPU GENERYCZNEGO W C++. 77

Spis tabel

TABELA 1: PRZYKŁADY DEKLARACJI ZMIENNYCH RESPEKTUJĄCYCH STYL NAZWY PO LEWEJ STRONIE I TYPU PO PRAWEJ DLA WYBRANYCH JĘZYKÓW 32
TABELA 2: PORÓWNANIE KONSTRUKCJI PROGRAMOWANIA FUNKCYJNEGO POD WZGLĘDEM ASPEKTÓW WIZUALNYCH Z C++20 ORAZ JAVY 8 71
TABELA 3: PORÓWNANIE KONSTRUKCJI PROGRAMOWANIA FUNKCYJNEGO POD WZGLĘDEM ASPEKTÓW FUNKCJONALNYCH Z C++20 ORAZ JAVY 8 71

Spis listingów

Listing 1: Przykład sformatowania pojedynczej linii kodu na kilka wierszy w tekście pracy. 8
Listing 2: Przykład konwencji sprecyzowania parametrów szablonowych w tej samej oraz w osobnej linii. 8
Listing 3: Sygnatury przeciążeń metody `sort()` dla list w języku Java 19. 12
Listing 4: Implementacja algorytmu sortowania bąbelkowego w Pythonie 3.10. 17
Listing 5: Implementacja algorytmu sortowania bąbelkowego tablicy w Javie w wersji 5+. 19
Listing 6: Implementacja algorytmu sortowania bąbelkowego wektora w C++ w wersji 11. 21
Listing 7: Definicja makr `MAX` oraz `MIN` w języku C i C++. 23
Listing 8: Przykład użycia makr `MAX` oraz `MIN`. 23
Listing 9: Przykład otrzymania błędnej wartości z powodu użyciu makra i preinkrementacji. 24
Listing 10: Rozwinięte makro `MAX` demonstrujące problem używania wyrażeń z efektami ubocznymi. 24
Listing 11: Przykład błędu kompilacji spowodowanego pominięciem inicjalizacji zmiennej podczas korzystania z `auto`. 29
Listing 12: Przykład użycia odpowiednio wyrównanego `std::array` jako bufora do danych. ... 29
Listing 13: Prezentacja tworzenia obiektu, którego inicjalizacja została celowo pominięta na tle domyślnie inicjalizowanych obiektów. 30
Listing 14: Deklaracja zmiennej bez widocznego konkretnego typu (użycie AAA). 31
Listing 15: Przykład użycia kodu generycznego ze wspieranym i niewspieranym typem w Pythonie.35
Listing 16: Treść błędu traktującego o niewspieranej operacji dla danego typu w Pythonie. 36
Listing 17: Przykład użycia kodu generycznego ze wspieranym i niewspieranym typem w Javie.... 36
Listing 18: Treść błędu traktującego o niewspieranej operacji dla danego typu w Javie. 37
Listing 19: Przykład użycia kodu generycznego ze wspieranym i niewspieranym typem w C++. 37
Listing 20: Treść błędu traktującego o niewspieranej operacji dla danego typu w C++ wyprodukowana przez kompilator GCC 13.1. 38
Listing 21: Treść błędu traktującego o niewspieranej operacji dla danego typu w C++ wyprodukowana przez kompilator Clang 16.0. 38
Listing 22: Treść błędu traktującego o niewspieranej operacji dla danego typu w C++ wyprodukowana przez kompilator MSVC. 39
Listing 23: Szablon funkcji `print()` 40
Listing 24: Próba przeciążenia szablonu `print()` dla „zwykłych” obiektów oraz dla zakresów. .. 40
Listing 25: Fragment błędu wygenerowanego na skutek niepoprawnej redefinicji szablonu `print()`. 41
Listing 26: Przykład użycia SFINAE bazującego na funkcji z określonym typem na końcu. 42
Listing 27: Przykład użycia Konceptów. 44
Listing 28: Przykład prezentujący jak Koncepty poprawnie współpracują z przeciążaniem szablonów. 44
Listing 29: Przykład alternatywnej składni definiowania argumentu generycznego. 45
Listing 30: Przykład Konceptów zdefiniowanych przez użytkownika. 46
Listing 31: Przykład użycia szablonu `addAllTo()`. 46
Listing 32: Implementacja szablonu `addAllTo()` za pomocą lokalnego warunku bazującego na wyrażeniu `requires()`. 47
Listing 33: Sygnatura i definicja fragmentu biblioteki standardowej odpowiedzialnego za działanie algorytmu `std::ranges::sort()` 48
Listing 34: Definicja Konceptu `sortable`. 49
Listing 35: Przykład użycia kodu generycznego ograniczonego Konceptami ze wspieranym i niewspieranym typem w C++. 50
Listing 36: Treść błędu kompilacji wygenerowana przez kompilator GCC 12.2.0 dla kodu z Listing 35. 50
Listing 37: Przykład użycia szablonu `std::sort()` 53
Listing 38: Przykład użycia pętli zakresowej (ang. *range-based `for()` loop*) w C++. 54
Listing 39: Przykład użycia pętli zakresowej (ang. *for-each loop / enhanced for statement*) w Javie.54
Listing 40: Przykład użycia pętli zakresowej (ang. *for loop*) w Pythonie. 55
Listing 41: Przykład implementacji funkcji weryfikującej, czy przekazany tekst jest palindromem. 56

Listing 42: Przykład użycia std::ranges::sort()... 56
Listing 43: Przykład użycia standardowego algorytmu sortowania wyłącznie na pierwszych trzech elementach wektora. 57
Listing 44: Hipotetyczne przeciążenie narzędzia sort()... 58
Listing 45: Sortowanie różnych typów z uwagi na ich pojedynczą cechę. 58
Listing 46: Przykład zastąpienia komparatorów projekcjami. 60
Listing 47: Generyczny algorytm bubbleSort() wspierający precyzowanie komparatora i projekcji. 60
Listing 48: Zastąpienie niektórych projekcji wskaźnikami do pól.. 61
Listing 49: Tożsama logika sortowania kolekcji z Listing 45 w języku Java..................................... 61
Listing 50: Logika sortowania z Listing 49 wykorzystująca metody-fabryki komparatorów. 62
Listing 51: Tożsama logika sortowania kolekcji z Listing 45 w języku Python. 62
Listing 52: Sortowanie kolekcji domyślnie nieporównywalnego typu przy pomocy std::sort() oraz komparatora... 63
Listing 53: Wyszukiwanie binarne w posortowanej kolekcji typów nieporównywalnych domyślnie przy pomocy klasycznego komparatora.. 63
Listing 54: Wyszukiwanie binarne w posortowanej kolekcji typów nieporównywalnych domyślnie przy pomocy heterogenicznego komparatora. 64
Listing 55: Sortowanie i wyszukiwanie binarne w kolekcji przy pomocy narzędzi zakresowych z C++20. 65
Listing 56: Proceduralne rozwiązanie problemu generowania wybranych potęg dwójki w języku Java. 66
Listing 57: Funkcyjne rozwiązanie problemu generowania wybranych potęg dwójki w języku Java.67
Listing 58: Koncept view, który musi spełniać każdy widok. 68
Listing 59: Zmienna szablonowa enable_view. 68
Listing 60: Przykład użycia adaptora std::views::take zwracającego widok na pierwsze trzy elementy wektora. 69
Listing 61: Aliasy przestrzeni nazw std::views oraz std::ranges. 69
Listing 62: Sortowanie pierwszych trzech elementów wektora przy pomocy sprecyzowania podzakresu dzięki adapterom. 70
Listing 63: Sortowanie trzech elementów z pominięciem pierwszego przy pomocy widoków drop oraz take. 70
Listing 64: Funkcyjne rozwiązanie problemu generowania wybranych potęg dwójki w C++20. 70
Listing 65: Powtórzony Listing 57.. 71
Listing 66: Implementacja algorytmu z Listing 41 z wykorzystaniem zakresów i widoków. 72
Listing 67: Warunkowe deklarowanie i definiowanie destruktora dzięki Konceptom.................... 75
Listing 68: Generyczna metoda przyjmująca dowolny typ, który implementuje dwa sprecyzowane interfejsy w Javie. 75
Listing 69: Generyczna metoda przyjmująca dowolny typ, który spełnia sprecyzowany Koncept oraz lokalne wymaganie w C++. 76
Listing 70: Usuwanie duplikatów z std::vector. 78
Listing 71: Usuwanie duplikatów z masters::vector. 81
Listing 72: Deklaracja i definicja szablonu masters::vector. 83
Listing 73: Brak możliwości dostępu do publicznego pola, które zostało odziedziczone prywatnie.84
Listing 74: Przykład użycia deklaracji using w celu upublicznienia prywatnie odziedziczonego pola. 84
Listing 75: Definicja makra GENERATE_CONTAINER_REQUIREMENTS_TYPES_FROM() automatyzującego generowanie kodu upubliczniającego wszystkie pola będące wspólne dla każdego kontenera.85
Listing 76: Szablon masters::vector wzbogacony o upublicznienie wspólnych dla wszystkich kontenerów pól. 85
Listing 77: Implementacja uzakresowionej wersji metody .erase(). 86
Listing 78: Przykład dodania elementów z kontenera std::string do kontenera std::vector. 87
Listing 79: Przykład dodania elementów z jednego wektora do drugiego, gdzie elementy są różnego typu, ale wspierają niejawną konwersję. 87
Listing 80: Przykłady użycia szablonu metody masters::vector::erase()....................... 87

Listing 81: Implementacja uzakresowionej wersji metody .insert(). 88
Listing 82: Implementacja Konceptu convertible_to_range_of. 88
Listing 83: Przykład użycia konstruktorów konwertujących całe zakresy. 89
Listing 84: Implementacja konstruktora konwertującego zakres dla masters::vector. 89
Listing 85: Przykład problemu ze stosowaniem szablonu konstruktora. 90
Listing 86: Konstrukcja kontenera na podstawie przeniesienia innego zakresu. 91
Listing 87: Przykład użycia operatorów porównujących dla typu std::vector. 92
Listing 88: Nagłówek szablonu operatora <=> dla masters::vector.................................... 93
Listing 89: Implamentacja szablonu operatora <=> dla masters::vector............................. 94
Listing 90: „Klasyczna” deklaracja obiektu typu std::vector<int>...................................... 96
Listing 91: Deklaracja obiektu typu std::vector<int> z pominięciem argumentu szablonu precyzującego typ elementu. 96
Listing 92: Implementacja wskazówek dedukcji argumentów szablonowych dla klas (CTAD). 97
Listing 93: Unikalny CTAD dla masters::vector. 97
Listing 94: Przykład użycia konstruktorów konwertujących całe zakresy z pominięciem argumentu szablonowego precyzującego typ elementu kolekcji. 98
Listing 95: Przykład automatycznie wygenerowanego CTAD dla konstruktora wykorzystującego std::initializer_list. 98