# Wzorzec Obserwator

Omawianie wzorców projektowych rozpocznę od wzorca obserwator. Wzorzec ten był dla mnie, w pewien sposób znaczący, w końcu w rzeczywistym świecie obserwowanie środowiska czy zachowań innych ludzi zajmuje znaczną część naszego życia. W końcu ludzie powiadamiają wszystkich wokół siebie o zmianie swojego stanu np. za pomocą emocji, ruchu czy słów. Również ciało ludzie jest skonstruowane w taki sposób że wszystkie receptory ciała powiadamiają mózg o zmianie swojego stanu, na podstawie których możemy podejmować różne reakcje.

Definicja wzorca obserwator brzmi – wzorzec definiuje pomiędzy obiektami relacje jeden do wielu w taki sposób, że kiedy wybrany obiekt zmienia swój stan, to wszystkie jego obiekty zależne zostają o tym powiadomione i automatycznie zaktualizowane.

Co oznacza to w praktyce? Wzorzec pozwala nam na zdefiniowanie takich związków pomiędzy obiektami, które komunikując się ze sobą będą przekazywały sobie informacje nie będąc zmuszonymi do wiedzy o swoich obszarach odpowiedzialności o ile implementują odpowiednie interfejsy.

Prześledźmy na przykładzie zachowanie wzorca obserwator.

Załóżmy że w jednostce wojskowej znajduje się oddział zwiadowców, których głównym zadaniem jest zbieranie informacji o sile i manewrach wroga. Zwiadowcy będą często działali blisko lub na terytorium wroga, więc będą musieli potrafić się kamuflować jak również będą musieli mieć możliwość szybkiego, pewnego i bezpiecznego przekazywania wszystkich informacji, które zdobędą. Ze względu na to że sytuacja na froncie jest dynamiczna, będą musieli mieć również możliwość informowania oddziałów bezpośrednio zainteresowanych zdobytymi przez nich informacjami, przy czym ilość informowanych oddziałów może się zmieniać w czasie (musi istnieć swoboda dodawania i usuwania obserwatorów).

// alternative

Nasz oddział powinien charakteryzować się następującymi cechami:

- jedyną informacja którą powinien wiedzieć o oddziałach które będą informowane(obiektach obserwujących) jest to że implementują one określony interfejs (interfejs Obserwator) w naszej alegorii możemy to porównać do posługiwania się określonym szyfrem, który będzie zrozumiały dla naszych zwiadowców i oddziałów które informują.

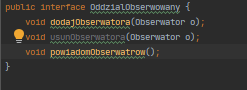
- nowe oddziały informowane (nowych obserwatorów) musimy dodawać i usuwać w dowolnym momencie – sytuacja na froncie potrafi być bardzo dynamiczna.

- dodatek nowych obserwatorów nigdy nie powinien pociągać za sobą konieczności modyfikacji obiektu obserwowanego (czyli naszego oddziału zwiadowców). Nie powinniśmy oczekiwać że nasz oddział nagle będzie musiał postawić most pontonowy dla kolumny czołgów – to nie jest jego zadanie.

- zarówno obiekty obserwowane jak i obserwujące mogą być od siebie niezależnie i wielokrotnie wykorzystywane – zawsze będzie mogła zaistnieć konieczność przegrupowania oddziałów i/lub manewrowania

- zmiany wprowadzone do obiektu obserwowanego jak i obiektów obserwujących nie mają na siebie wzajemnie żadnego wpływu (charakteryzują je tzw. luźne związki)– jeśli w naszym lub innych oddziałach zmieni się skład osobowy lub uzbrojenie to nie może mieć to wpływu na jego podstawową funkcjonalność – informowanie.

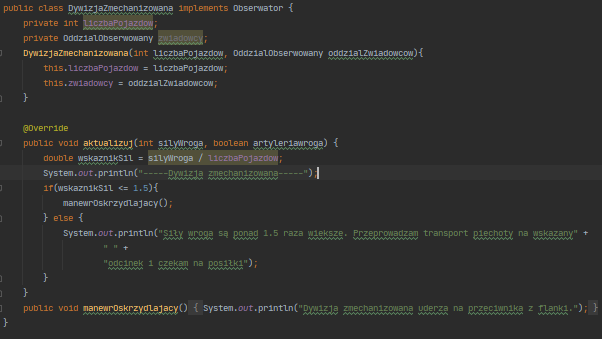
Na początku zdefiniujmy dwa interfejsy, które będą implementowane przez nasze klasy, interfejs obiektu obserwowanego i obserwatora. // najpierw oddział obserwowany i zwiadowcy a później obserwator i reszta

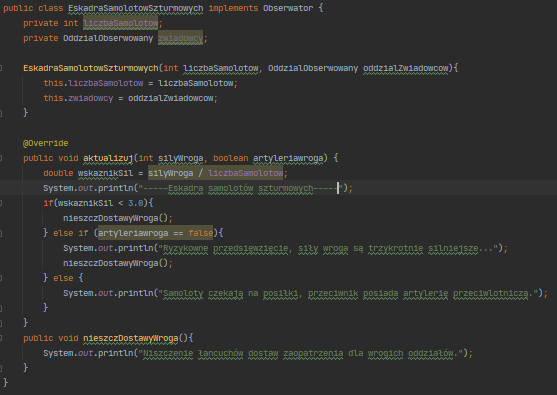


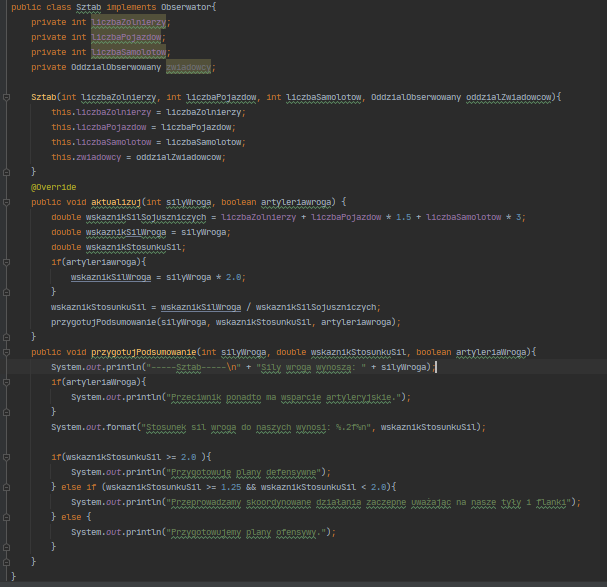
Następnie stwórzmy klasę zwiadowcy implementującą interfejs obiektu obserwowanego i implementujemy odpowiednie metody. Nasz obiekt obserwowany ma możliwość dodawania i usuwania obserwatorów. Klasa zawiera parametry opisujące liczbę żołnierzy w oddziale, liczbę żołnierzy wroga, którzy zostali zaobserwowani, oraz czy wróg ma wsparcie artyleryjskie. W metodzie zwiad(), losowo decydujemy czy napotkaliśmy wroga i jeśli tak to jakimi siłami dysponuje. Następnie informujemy o tym obserwatorów, przekazując te dane.



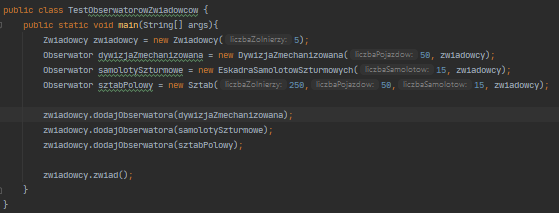
Prześledźmy teraz przykładowe klasy obserwujące: przygotowałem 3 proste klasy: DywizjaZmechanizowana, EskadraSamolotówSzturmowych i Sztab. We wszystkich tych klasach odbieram dane z obiektu obserwowanego(oddziału zwiadowców) i podejmuje decyzje co do zachowania moich obiektów a w klasie Sztab ponadto przygotowuję podsumowanie.



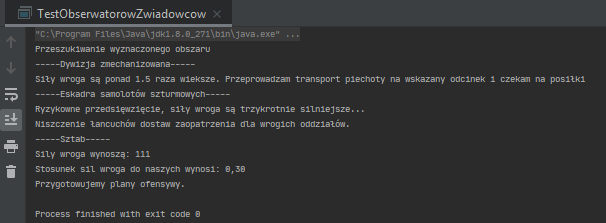




Ostatecznie przeprowadźmy test działania naszych zwiadowców. Tworzymy odpowiednie klasy i przekazujemy im dane w konstruktorach. Następnie dodajemy obserwatorów naszego oddziału i wyruszamy na zwiad.



Jeśli teraz nasi zwiadowcy zauważą oddziały wroga to przekażą te informacje do pozostałych oddziałów, a one zachowają się odpowiednio do stosunku sił.



Podsumowując: wzorzec obserwator jest wykorzystywany wszędzie tam gdzie konieczne jest automatyczne dostosowanie zachowań obiektów od zmiany stanu innych obiektów. Jest dość powszechny we wbudowanych bibliotekach Javy – posiada ona klasę Observable i interfejs Observer, ponadto jest szeroko wykorzystywany przy budowaniu GUI np. w ActionListenerach. Również komunikacja wewnątrz wzorca MVC pomiędzy widokiem a modelem opiera się na wzorcu Obserwator.

## Metoda szablonowa

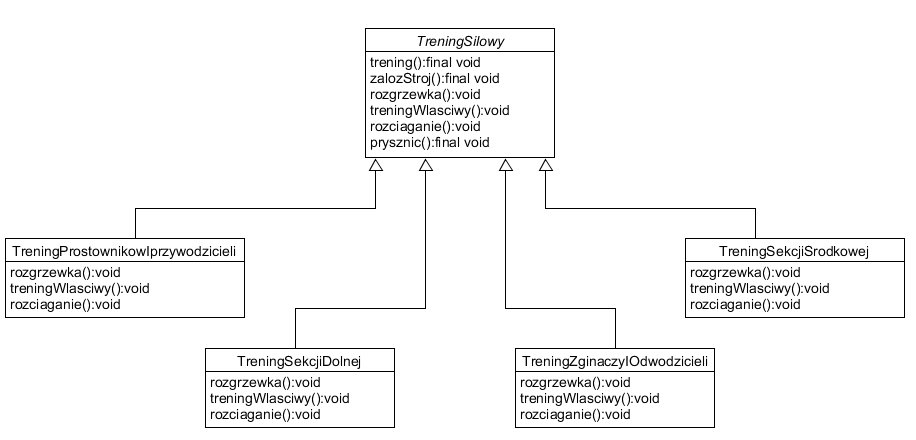
Następnym wzorcem, który chciałbym omówić jest wzorzec metoda szablonowa. Jak wiele innych wzorców wciela w życie zasadę o oddzieleniu i hermetyzacji zmieniających się elementów aplikacji. To co oddzielamy i hermetyzujemy w tym wzorcu to określony algorytm.

Definicja wzorca brzmi: wzorzec Metoda Szablonowa definiuje szkielet danego algorytmu w określonej metodzie, przekazując realizację niektórych jego kroków do klas podrzędnych. Wzorzec ten pozwala klasom podrzędnym na redefiniowanie pewnych kroków algorytmu, ale jednocześnie uniemożliwia zmianę jego struktury.

W praktyce oznacza to utworzenie metody z atrybutem final w obrębie nadrzędnej klasy abstrakcyjnej, którą będzie zawierała w sobie określone kroki algorytmu, które będą innymi metodami. Wszystkie klasy dziedziczące po tej klasie będą implementowały poszczególne kroki tej metody hermetyzującej algorytm. Jeśli przewidujemy że dany krok algorytmu będzie niezmienny to definiujemy go w klasie abstrakcyjnej.

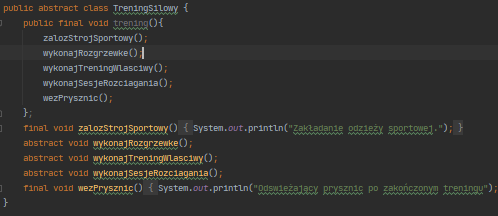
Rozpatrzmy następujący przykład: jako postanowienie noworoczne postanowiliśmy uczęszczać na regularne treningi siłowe. Mimo że liczba możliwych wariantów treningu (partii mięśniowych) jest znaczna to schemat treningu będzie wyglądał podobnie niezależnie od tego które mięśnie trenujemy w dany dzień. Zawsze będziemy musieli ubrać się w strój sportowy, przeprowadzić rozgrzewkę, następnie przystąpić do właściwego treningu, po czym porozciągać trenowane grupy mięśniowe, kończąc trening odświeżającym prysznicem.

Diagram klas dla tego przypadku będzie wyglądał następująco:

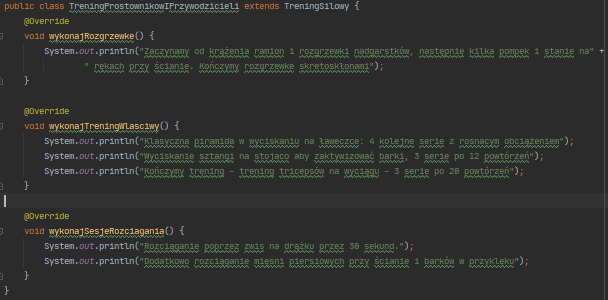


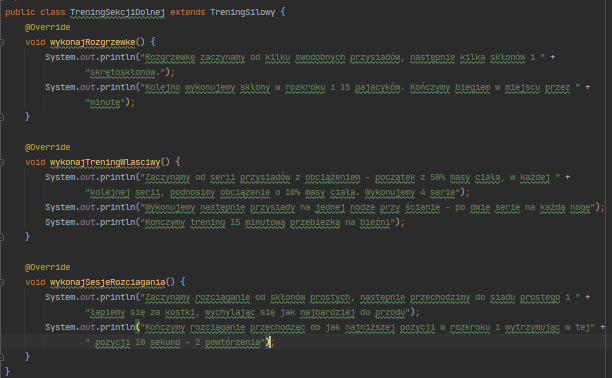
Klasa abstrakcyjna TreningSiłowy definiuje metodę trening zawierającą szablon algorytmu. Definiuje również metody składające się na algorytm – metody które nie będą się zmieniać w poszczególnych klasach zostają zaimplementowane już w klasie TreningSilowy.

Popatrzmy na kod:

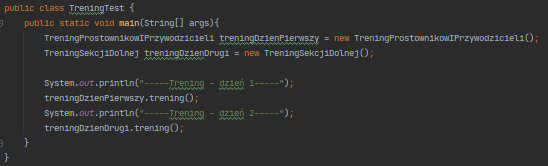


Klasy rzeczywiste natomiast implementują zdefiniowane metody abstrakcyjne:

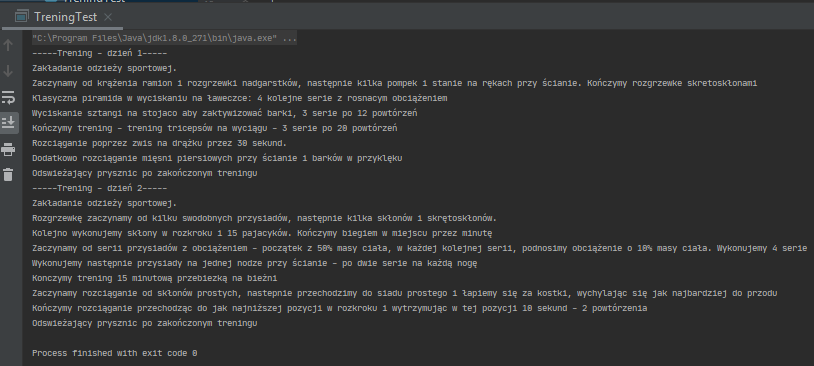




Przetestujmy teraz nasze plany treningowe:



W konsoli powinniśmy otrzymać następujący wynik:



Jak widać poszczególne kroki treningu są takie same mimo iż różnią się implementacją.

Podsumowując wzorzec jest wykorzystywany wszędzie tam gdzie konieczne jest określenie niezmiennego szablonu zachowań, który jednak można dostosowywać do swoich potrzeb. Jest całkiem szeroko wykorzystywany w bibliotekach języka Java takich jak Java.io.InputStream, Java.io.OutputStream czy Java.util.AbstractList.

# Wzorzec budowniczy

W czasie poszukiwania materiałów do wpisu na temat wzorca budowniczy spotykałem się najczęściej z dwoma wersjami tego wzorca: klasycznym podejściem GoF oraz tzw. Static Builder. We wpisie spróbuje krótko przedstawić oba.

Niezależnie od używanej wersji definicja jest jedna – jest to wzorzec umożliwiający budowanie złożonych obiektów krok po kroku, używając określonego algorytmu. logika mówiąca o tym jak obiekt ma być zbudowany **będzie oddzielona od implementacji** tej logiki. Dodatkowo w klasycznej wersji GoF proces tworzenia obiektu jest kontrolowany przed dodatkowy obiekt klasy *Director,* który kontroluje cały proces.

Wzorzec ma wiele wspólnego ze wzorcem Fabryka abstrakcyjna i czasem może wyglądać bardzo podobnie. Różnicą między nimi jest to że wzorzec fabryka abstrakcyjna koncentruje się na tym **jaki** produkt ma być utworzony a wzorzec budowniczy na tym **jak** to zrobić.

Zwykle wzorca budowniczy używamy gdy:

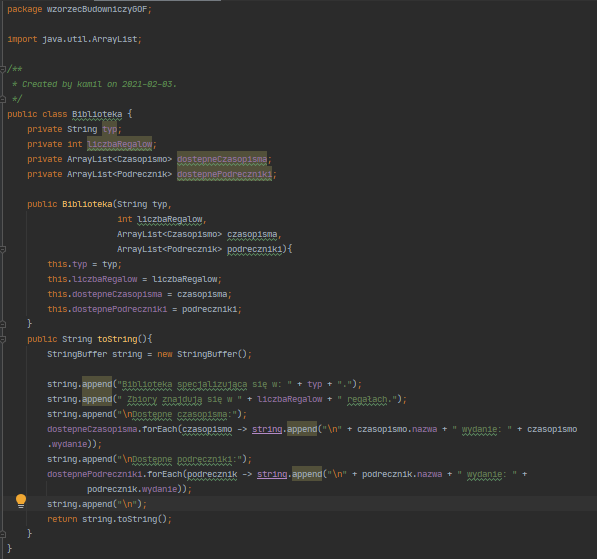
- obiekt, który chcemy utworzyć jest złożony i nie możemy go uprościć

- nie da się stworzyć egzemplarza obiektu od razu, a proces budowania obiektu wymaga kilku określonych operacji.

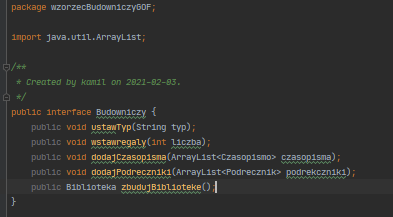
- tworzony obiekt będzie jednym z większego zestawu obiektów, które będą budowane według jednego schematu, ale na różne sposoby.

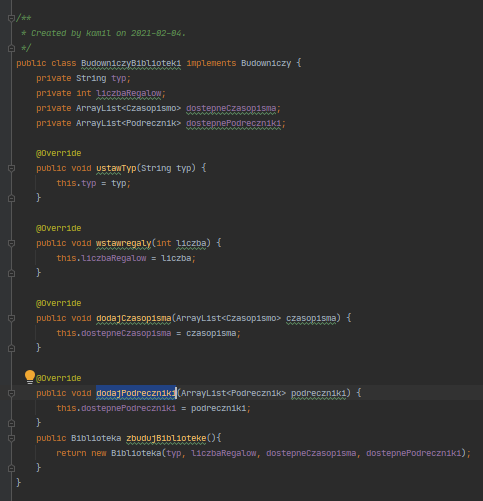
Rozpatrzmy teraz prosty przykład – użyjemy wzorca Budowniczy, do zbudowania w programie obiektu reprezentującego Bibliotekę.

Zacznijmy od klas biblioteki. Zawiera pola określające jej typ (np. akademicka, motoryzacyjna itp.), liczbą regałów, oraz listę czasopism oraz podręczników.

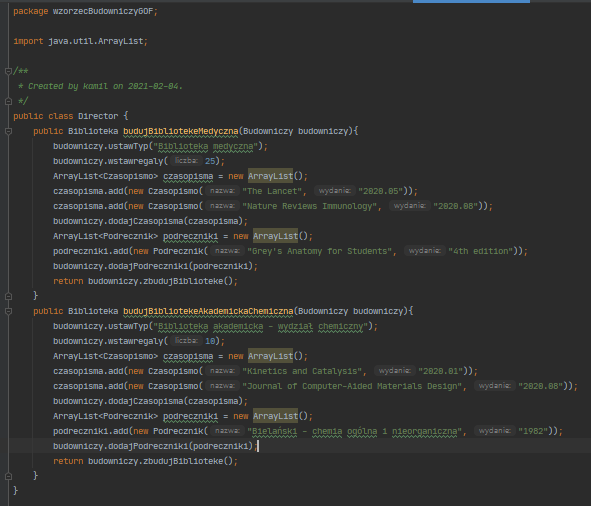


Następnie interfejs budowniczy zapewniający metody służące inicjalizowaniu poszczególnych pól obiektu i klasa rzeczywista implementująca ten interfejs:

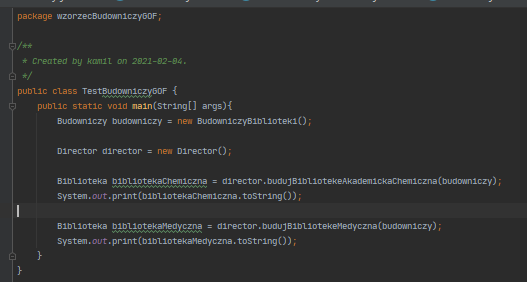




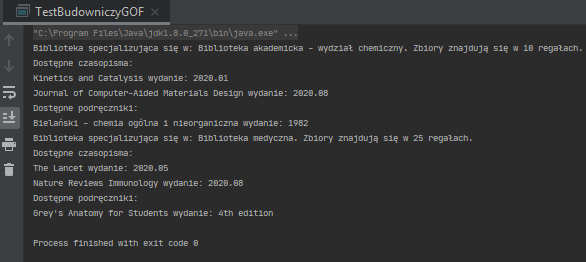
Kolejno klasa direktor – zarządzająca procesem budowy obiektu – budujemy nasze złożone obiekty wieloetapowo. Ponadto tworzone obiekty mimo że są jednej klasy to budowane są na różne sposoby.



Teraz przetestujmy naszego budowniczego:



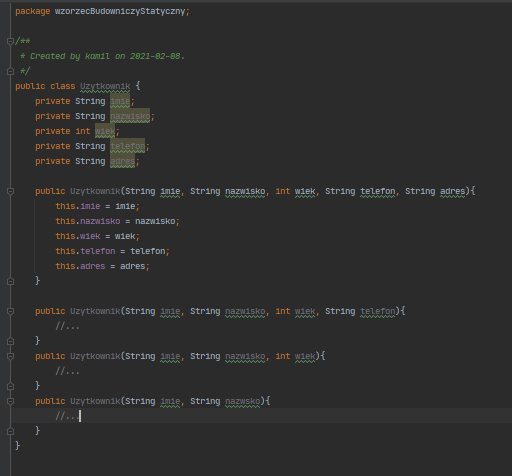
Ostatecznie w konsoli powinniśmy otrzymać reprezentacje zbudowanych obiektów:



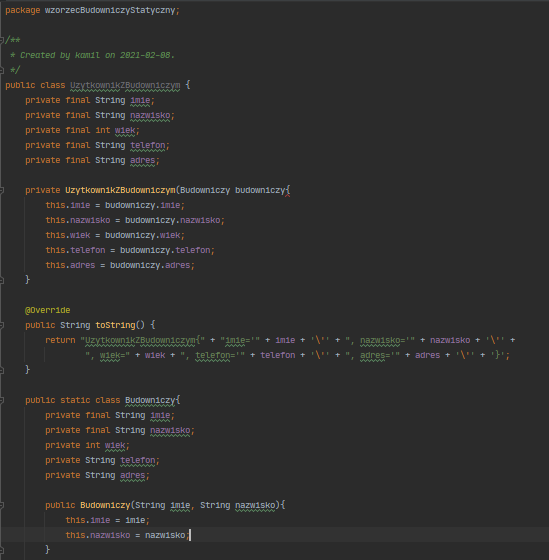
Static builder

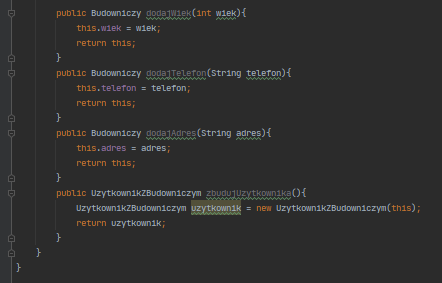
Mimo że statyczny budowniczy nie jest oficjalnym wzorcem projektowym, w wielu stronach opisujących wzorzec znajdzie się właśnie taki przykład. Ta konstrukcja, po raz pierwszy zaproponowana przez Josha Blocha, wykorzystuje do budowania obiektów tzw. fluent interface. Podejście to zakłada wykorzystanie metod zwracających instancję klasy w obrębie której się znajdują. Dzięki takiemu podejściu można wywoływać te metody jedna po drugiej w kaskadzie, z których każda reprezentuje etap budowania obiektu.

Rozpatrzmy prosty przykład. Załóżmy że potrzebujemy klasy reprezentującej użytkownika, której obiekt po inicjalizacji będzie niemutowalny tzn. nie będziemy mogli zmieniać jego stanu a jego wszystkie pola będą prywatne. Obiekt będzie miał takie atrybuty jak imię, nazwisko, wiek, telefon oraz adres. Aby utworzyć obiekt niemutowany wszystkie te informacje musielibyśmy przekazać w konstruktorze co było by mało czytelne. Ponadto jeśli tylko imię i nazwisko byłyby polami wymaganymi a pozostałe opcjonalnymi musielibyśmy przeładować konstruktor z różnymi przyjmowanymi atrybutami.

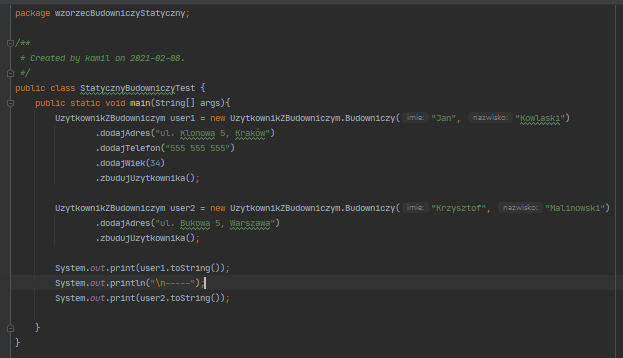


Tutaj z pomocą przychodzi nam statyczny budowniczy. Za jego pomocą możemy zachować niemutowalność zbudowanego obiekt, a sam proces konstrukcji staje się elastyczny i łatwy w rozbudowie.

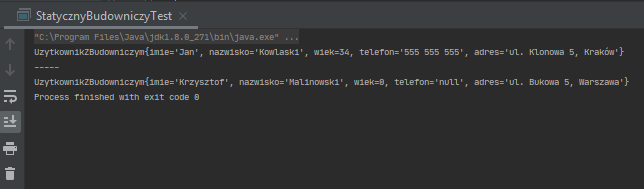




Przetestujmy teraz naszego budowniczego



Ostatecznie w konsoli powinniśmy otrzymać:



Jak widać kod jest dużo bardziej czytelny jak i elastyczny. Jednocześnie cały czas mamy zapewnioną niemutowalność obiektu.

Wzorzec budowniczy jest często wykorzystywany w języku Java. Wykorzystuje się go w takich miejscach jak np. StringBuilder, czy StringBuffer. Do korzyści zastosowania wzorca można wymienić:

- oddzielenie kodu konstrukcyjnego od logiki biznesowej (zasada pojedynczej odpowiedzialności)

- konstruowanie obiektu etapami, z których nie wszystkie muszą być wykonywane od razu

- używanie tego samego kodu do budowania kolejnych reprezentacji produktów

- daje doskonałą kontrolę nad etapami budowania kompozytu

Do wad wzorca można zaliczyć znaczące rozbudowanie kodu i dodawanie sporej ilości klas.