ACTA UNIVERSITATIS LODZIENSIS
FOLIA OECONOMICA 211, 2007

*Paweł Rościzak**

# IMPLEMENTACJA SILNIKA SZTUCZNEJ SIECI NEURONOWEJ PRZY UŻYCIU WZORCÓW PROJEKTOWYCH W JĘZYKU JAVA

## 1. WPROWADZENIE

Zagadnienia ekonomiczne, obok problemów technicznych, są głównym obszarem wykorzystywania obliczeń neuronowych. Przez długi czas najpopularniejszą techniką badania i odzwierciedlania zjawisk gospodarczych było modelowanie liniowe. Zjawiska społeczno-ekonomiczne należą jednak do wyjątkowo skomplikowanych i niejawnych, a procesy zachodzące w gospodarce mają z reguły charakter wysoce nieliniowy i trudny do przedstawienia w postaci równań matematycznych. Modelowanie zależności ekonomicznych, wydaje się więc naturalnym obszarem zastosowania sztucznych sieci neuronowych, których właściwości powinny doskonale odpowiadać opisanym warunkom.

Korzystanie ze sztucznych sieci neuronowych pociąga za sobą wykonywanie ogromnej liczby złożonych i często powtarzających się operacji matematycznych. Nakład obliczeń potrzebny do nauczenia sieci, praktycznie wyklucza możliwość rozwiązania problemu za pomocą kartki oraz ołówka i wymusza skorzystanie z narzędzi o potężnych mocach obliczeniowych. Rozwój badań nad systemami neuronowymi i wprowadzenie ich do praktycznych zastosowań były więc silnie związane z pojawianiem się i upowszechnieniem komputerów osobistych oraz efektywnych sposobów ich programowania. Spełnienie wspomnianych warunków przyczyniło się do gwałtownego wzrostu zainteresowania oraz rozkwitu badań nad obliczeniami neuronowymi, który zainicjowany od połowy lat 80. XX w. trwa praktycznie do dziś.

Obecnie istnieje wiele wyspecjalizowanych pakietów komputerowych implementujących całą paletę najróżniejszych odmian sieci neuronowych oraz

* Mgr, pracownik w Zakładzie Informatyki Ekonomicznej Uniwersytetu Łódzkiego.

algorytmów ich uczenia. Do powszechniej znanych możemy zaliczyć np. Brain, BrainMaker, CascadeCorrelation, Dana, Explorer, Explornet, MathlabToolbox, Nestor, Neudisk, NeuralWorks, NeuroSolution, NuWeb, Propagator, StatisticaNeuralNetworks. Obok wymienionych wcześniej istnieją również dziesiątki innych, komercyjnych komputerowych systemów neuronowych oraz niezliczona liczba aplikacji darmowych tworzonych w różnych językach programowania i udostępnianych w Internecie[1].

Wspomniane pakiety doskonale nadają się do pełnienia funkcji narzędzi wszędzie tam, gdzie istnieje potrzeba użycia systemów neuronowych bez potrzeby wnikania w szczegóły wykorzystanych algorytmów. Z odmienną sytuacją mamy jednak do czynienia w przypadku, gdy przedmiotem zainteresowania są procesy przetwarzania oraz algorytmy uczenia. Programy stworzone przez innych nie dają wówczas pełnej swobody modyfikacji i testowania interesujących nas szczegółów i właściwości, a jedynym sposobem na pełną dowolność w implementowaniu struktury i algorytmów jest samodzielne oprogramowanie systemu sieci neuronowej oraz metod jej uczenia.

## 2. JĘZYK JAVA

Upowszechnienie się idei programowania zorientowanego obiektowo (ang. *Object Oriented Programming* – OOP) było kolejnym krokiem do przyspieszenia oraz usprawnienia procesu tworzenia i rozwoju nowego oprogramowania. Języki zorientowane obiektowo, w odróżnieniu od wcześniej wykorzystywanego podejścia proceduralnego, eliminują ograniczenia jednorodnej przestrzeni nazw oraz umożliwiają projektowanie i implementowanie konkretnych problemów w sposób bliższy ludzkiemu sposobowi rozumowania.

Java jest językiem całkowicie opartym na koncepcji programowania zorientowanego obiektowo. Każda operacja i każdy zbiór danych musi stanowić część składową obiektu lub pojęcia obejmującego obiekty tego samego typu, czyli klasy. Hermetyzacja pól i metod oraz dziedziczenie i polimorfizm, czyli podstawowe atrybuty obiektowości, pozwalają na wielokrotne wykorzystanie kodu, a wbudowane mechanizmy obsługi błędów za pomocą wyjątków, identyfikacji typu w czasie wykonania oraz wielowątkowość dostarczają dalszych, nowoczesnych i użytecznych możliwości programistycznych. Java jest jednak przede wszystkim językiem programowania sieciowego, należącym do kategorii języków wysokiego poziomu. Przyglądając się jej podstawowym ideom widać wyraźne dążenie twórców do uproszczenia

[1] Obszerny wykaz i omówienie oprogramowania neuronowego można znaleźć na stronach: Neural Network FAQ (newsgroup comp.ai.neural-nets), ed. W. S. Sarle, SAS Institute, 1997–2002, ftp://ftp.sas.com/pub/neural/FAQ5.html#questions, 01.07.2003 r.

i skrócenia procesu programowania. Cały szereg problemów związanych ze zwalnianiem zmiennych, wychodzeniem poza rozmiar tablicy, z jakimi spotyka się programista np. C++, został przerzucony na mechanizmy wewnętrzne języka. Implementacja większości typowych zadań staje się dzięki temu szybsza i prostsza, ponieważ programistą może bardziej skoncentrować się na poszukiwaniu rozwiązania problemu i poświęcić mniej uwagi na unikanie niebezpieczeństw wynikających z niuansów języka[2]. Java pozwala jednocześnie zachować całkowitą kontrolę nad kodem, nie posuwając się do uzależnienia od środowisk typu RAD (ang. *Rapid Application Development*) i nie wydzielając programiście miejsc, które mogą podlegać jego nieskrępowanej inwencji twórczej.

Trudno jednak uznać Javę za język optymalny we wszystkich dziedzinach. To co chroni programistę przed popełnianiem błędów, jest jednocześnie ograniczeniem, które nie pozwala zejść na niższy poziom programowania, np. w API (ang. *Aplication Programming Interface*) systemu operacyjnego. Programy w Javie są niezależne od konkretnej platformy, co zostało osiągnięte przez wprowadzenie elementu pośredniczącego pomiędzy aplikacją a systemem operacyjnym. Tym elementem jest interpreter określany nazwą Wirtualnej Maszyny Javy (ang. *Java Virtual Machine*). Program wykonywany w ten sposób nie ma szans na osiągniecie wydajności programów skompilowanych do postaci kodu maszynowego właściwego dla konkretnej platformy. Wspomniane cechy, będące źródłem doskonałych właściwości sieciowych Javy, sprawiają jednocześnie, że przegrywa ona na wielu polach z językami takimi, jak C++, pozwalającymi na programowanie względnie niskopoziomowe, dużą kontrolę nad kodem i pozbawionymi elementu interpretera.

Implementacja sieci neuronowej oraz algorytmów jej uczenia jest zagadnieniem, które z powodzeniem może zostać oprogramowane w sposób wykorzystujący standardowe rozwiązania, dostarczane przez mechanizmy języka Java. Drugim silnym atutem skłaniającym do wykorzystania Javy są jej ogromne możliwości sieciowe, które otwierają interesujące perspektywy zarówno prezentacji w Internecie, jak i skorzystania z zalet programowania rozproszonego. Do wspomnianych zalet należy również używanie przez Javę międzynarodowego, 16-bitowego standardu znakowego o nazwie Unikod. Stosując Unikod mamy gwarancję, że po zapisaniu dowolnego znaku na jednym systemie operacyjnym, na innych platformach uzyskamy dokładnie taki sam znak[3].

---

[2] B. Eckel, *Thinking in Java. Edycja polska. Wprowadzenie do programowania zorientowanego obiektowo w sieci WWW*, Wydawnictwo Helion, Gliwice 2001, s. 21–24.

[3] Ta bardzo użyteczna zasada nie obowiązuje jednak w przypadku JVM pracującej na konsoli systemów operacyjnych Windows. Więcej informacji na temat Unikodu można znaleźć w serwisie Komitetu Technicznego Unikodu (Unicode Technical Committee–ETC), zob. *Czym jest Unikod?*, http://www.unicode.org/standard/translations/polish.html, 01.09.2003 r.

Właściwości wynikające z programowania sieciowego i rozproszonego, połączone z możliwością wielokrotnego wykorzystania już istniejących klas, pozwalają względnie łatwo obudować implementacje obliczeń neuronowych do postaci potężnych sieciowych systemów informatycznych. Użycie apletu do prezentacji działania już nauczonej sieci eliminuje konieczność instalowania oprogramowania[4], a użytkownik ma jednocześnie pewność, że aplikacja nie może wyrządzić żadnych szkód na jego komputerze[5]. Wykorzystanie modelu klient–serwer, np. do zdalnego uczenia i przetwarzania danych przez system neuronowy, również nie stanowi większego problemu w Javie. Wiele zadań, takich jak np. optymalizacja struktury dużych sieci neuronowych za pomocą algorytmów genetycznych, wymaga potężnych mocy obliczeniowych. Doskonałym rozwiązaniem może być w takiej sytuacji zastosowanie koncepcji masowych obliczeń równoległych, wymagających także języka o dużych możliwościach programowania sieciowego i dysponujących już przykładami implementacji w Javie[6].

## 3. WZORCE PROJEKTOWE W JAVIE

Podczas tworzenia aplikacji programista rozwiązuje szereg konkretnych problemów, z których duża część jest podobna lub nawet taka sama dla większości programów komputerowych. Mamy więc do czynienia z pewnymi powtarzającymi się rozwiązaniami, które doskonale nadają się do ujęcia w bardziej ogólne wzorce, łączące w sobie najlepsze ze znanych rozwiązań i tworzące rodzaj modelowej implementacji konkretnego problemu. Potrzeba opracowania takich schematów jest tym bardziej uzasadniona w odniesieniu do programowania zorientowanego obiektowego, które z założenia ma umożliwiać wielokrotne wykorzystanie kodu oraz operowanie na klasach i obiektach.

Wymienione powody legły u podstaw upowszechnienia się koncepcji i pojęcia wzorca projektowego (ang. *design pattern*), omówionej w fundamentalnej pozycji autorstwa tzw. Gangu Czworga (ang. *Gang of Four*) pt. *Wzorce projektowe*[7]. Pojęcie wzorca projektowego opisuje strukturę bardziej ogólną od klasy i odnosi się do opisu strategii wzajemnych zależności i zasad komunikacji pomiędzy obiektami. Zgodnie z jedną z popularniejszych definicji:

[4] Zob. np. aplety sieci neuronowych wykonane przez autora i umieszczone na stronach http://pawelrosczak.republika.pl/emil/ oraz http://pawelrosczak.republika.pl/makrosim/, 01.03.2005 r.

[5] E. J. Harold, *Java. Programowania sieciowe*, Wydawnictwo READ ME, Warszawa 2001, s. 73–75.

[6] *Ibidem*, s. 9–10.

[7] Zob. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison Wesley Professional, Reading 1994.

„Wzorce projektowe stanowią powtarzalne rozwiązanie zagadnień projektowych, z którymi się wciąż spotykamy"[8].

Wprowadzenie wzorców projektowych jest najefektywniejszym rozwiązaniem w sytuacji, gdy jeden i ten sam problem musi rozwiązywać wielu programistów w różnych projektach. Wymyślanie rozwiązania przez każdego z nich osobno i od początku, nie tylko wydłuża proces tworzenia, ale praktycznie zawsze prowadzi do rozwiązania mniej optymalnego. Skorzystanie ze wzorca, będącego dopracowanym efektem pomysłów wielu programistów, nawet jako punktu wyjścia do dalszych prac, skraca czas tworzenia i podnosi jakość efektu końcowego. Rozpowszechnianie się wzorców oznacza jednocześnie łatwość korzystania z cudzych fragmentów kodu, które, o ile wpisują się w ramy znanych schematów, nie stanowią już przedmiotu do zgłębiania, ale standardowe cegiełki do sprawnego budowania większej całości. Wzorce to wreszcie rozwój fachowej terminologii, która umożliwia zwięzłą i precyzyjną komunikację oraz tak samo lakoniczne i jednoznaczne opisy.

## 4. KONCEPCJA SILNIKA NEURONOWEGO

Punktem wyjścia implementacji sieci neuronowej i algorytmów uczenia było dążenie do zbudowania jak najbardziej uniwersalnej aplikacji, pozwalającej przy wykorzystaniu parametrów oraz systemu plików przeprowadzać obliczenia, a także trening na możliwie szerokiej palecie dowolnych, jednokierunkowych, wielowarstwowych sieci neuronowych z sigmoidalną funkcją aktywacji neuronu[9]. Główna idea projektu koncentruje się na stworzeniu oprogramowania obejmującego przede wszystkim istotę obliczeń neuronowych, będącego swego rodzaju silnikiem, który z powodzeniem można wykorzystać samodzielnie lub łatwo włączyć do współpracy z innymi aplikacjami.

Omawiany program komputerowy, nazwany roboczo NeuralEngine, został w całości napisany w języku Java, przy użyciu API zgodnego ze środowiskiem wykonywania J2SE v1.2.2 (ang. Java 2 Platform, Standard Edition, version 1.2.2) lub późniejszym[10]. W celu maksymalnego rozszerzenia możliwości wielokrotnego wykorzystania kodu oraz ułatwienia konserwacji i rozwijania programu, jego struktura ogólna została ujęta, wszędzie tam, gdzie było to uzasadnione, w postaci standardowych wzorców projektowych. Obok dążenia

[8] Cyt. za J. W. Cooper, *Java. Wzorce projektowe*, Wydawnictwo Helion, Gliwice 2001, s. 17.

[9] Więcej na temat implementowanego rodzaju sieci neuronowej oraz algorytmów jej uczenia zob. P. Rościzak, *Implementacja i wykorzystanie wielowarstwowej sieci perceptronowej w modelowaniu makroekonomicznym*, http://pawelrosczak.republika.pl/mlp/, 01.03.2005 r.

[10] Więcej na temat platformy J2SE zob. *Java 2 Platform, Standard Edition (J2SE)*, http://java.sun.com/j2se/, 01.09.2003 r.

do elastyczności na poziomie kodu, program koncentruje się na potencjalnym umożliwieniu współpracy z oddzielnie tworzonymi aplikacjami na poziomie funkcjonalności systemu operacyjnego.

Istnieją trzy sposoby komunikowania się z aplikacją. Wszystkie korzystają z opracowanego na potrzeby obliczeń neuronowych zbioru komend[11], określających rodzaj żądanej operacji oraz parametry jej wykonania. Program nie ma wpisanej na stałe do kodu struktury sieci neuronowej, ale korzysta z plików tekstowych o ustalonym formacie. Podobnie został rozwiązany problem budowy sieci, obserwacji i danych wejściowych. Wszędzie tam, gdzie jest to wymagane, informacje również są wczytywane z plików tekstowych o zdefiniowanej strukturze[12]. Takie same rodzaje plików, jak w przypadku danych uczących, służą do zapisywania efektów wykonywanych obliczeń, a wszystkie wymienione informacje przekazywane są do aplikacji jako nazwy odpowiednich plików w postaci parametrów wywoływanych poleceń.

Pierwszą metodą pracy z aplikacją jest wpisanie odpowiedniej komendy bezpośrednio w wierszu poleceń. Wyniki operacji zawsze zapisywane są do wskazanych plików oraz wysyłane na standardowe wyjście, czyli w tym przypadku na konsolę. Drugim, o wiele bardziej efektywnym, sposobem wykonywania nawet całej serii operacji jest wpisanie polecenia wczytania i wykonania pliku tekstowego z zestawem komend. Komendy w skrypcie muszą być zakończone średnikiem (;), a sam tekst może zawierać znane, np. z języków C, C++ lub Java, komentarze jednowierszowe, zaczynające się od znaków „//", a wygasające z końcem wiersza.

```
randomize weights using uniform distribution
for network weights_8_6_4_0.txt
send result into weights_8_6_4_0.txt
with min -0.5 // komentarz jednowierszowy
max 0.5;
```

Przykład 1. Plik z przykładowymi poleceniami programu NeuralEngine v1.0

Włączenie do programu możliwości przetwarzania wsadowego opartego na plikach z poleceniami pozwala skrócić do minimum interakcję z interfejsem. Użytkownik może wykorzystać gotowe schematy poleceń, a następnie za pomocą jednej krótkiej komendy `execute nazwaPliku` błyskawicznie zlecić aplikacji wykonanie całego szeregu skomplikowanych operacji.

[11] Opis poleceń i ich parametrów znajduje się we fragmencie pt. *Składnia poleceń programu NeuralEngine v1.0.*, w tekście P. Rościak, *Implementacja...*, s. 125–130.

[12] Struktura plików z zapisem informacji o sieci neuronowej opisana jest we fragmencie pt. *Struktura plików wykorzystywanych przez program NeuralEngine v1.0.*, w tekście P. Rościak, *Implementacja...*, s. 130–132.

Wykorzystywanie takich rozwiązań w dobie popularności GUI (ang. *Graphical User Interface*), czyli popularnych okienek i formularzy, może wydawać się nieco archaiczne i skomplikowane. Jednak praktyka użytkowania, wykraczająca poza chwilowy kontakt z aplikacją, wskazuje na ogromną przewagę przetwarzania wsadowego. Łatwo wyobrazić sobie korzystanie z odpowiednio dopasowanego do problemu interfejsu GUI przy trenowaniu sieci. Przede wszystkim należałoby wówczas otworzyć minimum dwa pliki, jeden zawierający strukturę sieci oraz drugi z obserwacjami. W przypadku argorytmu wczesnego stopu w grę wchodziłoby nawet więcej plików. Dodatkowo należałoby wskazać ścieżkę zapisu wyników oraz wpisać w odpowiednie miejsca formularza pozostałe parametry liczbowe algorytmu. Liczba operacji, jakie trzeba wykonać na oknach GUI, jest całkiem spora, a w przypadku całej sekwencji podobnych lub powiazanych poleceń, staje się ona długą i nużącą rytyną, którą można zastąpić wpisaniem w konsoli jednego krótkiego polecenia.

Mimo powyższych argumentów, brak interfejsu graficznego jest znaczącym ograniczeniem, które jednak można łatwo wyeliminować, wykorzystując możliwości trzeciego sposobu komunikacji z aplikacją. Każda komenda, którą wpisuje się w wierszu poleceń, może być także parametrem wykonania programu. W takim przypadku system wykonuje przekazane polecenie, po czym kończy działanie pozostawiając wyniki we wskazanych w komendzie plikach.

```
java -jar NeuralEngine10.jar error for network
aebp_weights_8_6_4_0.txt data data.txt send result
into error.txt
```

Przykład 2. Wywołanie aplikacji NeuralEngine v1.0 w formie pliku jar z komendą w postaci parametru

Można więc wyobrazić sobie program GUI będący nakładką graficzną do silnika neuronowego i korzystający z wywoływania tej drugiej aplikacji z parametrami lub przechwytujący jej standardowe wejście i wyjścia. Rozwiązanie takie można posunąć jeszcze dalej i wykorzystując doskonałe możliwości sieciowe Javy, przebudować silnik neuronowy także do postaci serwera. W tym momencie mamy do czynienia ze strukturą aplikacji rozproszonej, której dodatkową zaletą jest niezależność poszczególnych elementów od siebie. Części składowe systemu mogą być napisane w różnych językach programowania i modyfikowane oraz zamieniane bez konieczności korzystania z kodu bądź nawet kompilacji drugiego programu. Jako przykład takiego oprogramowania można zaprezentować system Ghostscript, będący konsolową przeglądarką plików PostScript, PDF i innych. Ghostscript może

działać samodzielnie lub zostać rozszerzony o nakładkę graficzną o nazwie GSview, będącą osobną aplikacją i podlegającą osobnej instalacji[13].

## 5. INTERPRETER POLECEŃ I KONCEPCJA ŁAŃCUCHA ODPOWIEDZIALNOŚCI

Interpreter poleceń jest, obok pakietu klas implementujących obliczenia neuronowe, główną częścią prezentowanego programu. Wzorzec interpretera stosowany jest przede wszystkim w zadaniach wymagających przetwarzania równań matematycznych, generowania zróżnicowanych i trudnych do ujednolicenia wyników oraz, tak jak w poniższym przykładzie, przy konieczności wykonywania podobnych lub powtarzających się poleceń użytkownika[14]. Interfejs takiego wzorca musi obejmować funkcje przeprowadzające – rozbiór komendy na części składowe, wyodrębnienie parametrów, utworzenie odpowiedniego obiektu implementującego wzorzec polecenia i wywołanie metody rozpoczynającej jego wykonanie.

W omawianym przykładzie występują dwie funkcje różniące się od siebie rodzajem parametru. Do pierwszej z nich przekazywany jest łańcuch tekstowy, co jest naturalne podczas interpretacji poleceń tekstowych, a do drugiej tablica parametrów tekstowych, co z kolei może być wygodniejsze przy bardziej zaawansowanym procesie rozbioru komendy.

Interpretowanie przebiega w nieco odmienny sposób w przypadku wykorzystywania funkcji i poleceń. Różnic jest wiele, od kwestii wartości zwracanej do sposobu grupowania i oddzielania parametrów, a wszystkie one skłaniają do utworzenia pierwszej klasy w hierarchii dziedziczenia, wyspecjalizowanej w tym wypadku pod kątem poleceń.

Interpretowanie rozpoczyna się w funkcji `public void interpret(String text)` od przekształcenia łańcucha tekstowego na tablicę słów oddzielonych w pierwotnym poleceniu tzw. ogranicznikami (ang. *delimiters*), czyli spacją, tabulatorem lub znakiem nowego wiersza. Klasa dostarcza ponadto zbioru funkcji statycznych, implementujących typowe operacje, jakie będą wykonywane w klasach konkretnych, a związane z usuwaniem słów nie mających znaczenia w poleceniu, czyli pełniących jedynie funkcję ozdobnika – `protected static String [] removeDecoration(String [] parameters, String [] decoration)`, usuwaniem pierwszego członu polecenia – `protected static String [] removeFirst(String [] parameters)` i grupowaniem parametrów niezbędnych

[13] Więcej na temat Ghostscript i Gsview zob. *Ghostscript, Ghostview and GSview*, http://www.cs.wisc.edu/~ghost/, 01.03.2003 r.

[14] J. W. Cooper, *Java...*, s. 157–158.

do utworzenia obiektu implementującego polecenie – `protected static Map groupArguments(String[] parameters, String[] names)`. Wymienione funkcje muszą mieć formę statyczną, ponieważ klasy konkretnych interpreterów mają własne, wspólne dla całego polecenia, statyczne tablice przekazywane do nich w formie parametrów. We wspomnianych tablicach przechowywane są wzory nazw argumentów komendy oraz wzory słów pełniących funkcję ozdobnika. Jedyną funkcją jaką należy zaimplementować w konkretnych klasach interpreterów jest `public abstract void interpret(String[] parameters)`.

Główna idea interpretowania odwołuje się do wzorca o nazwie łańcuch odpowiedzialności (ang. *chain of responsibility*). Jego działanie sprowadza się do umożliwienia pewnej liczbie połączonych ze sobą łańcuchowo obiektów podjęcia próby obsłużenia żądania. Żadna z klas nie wie o możliwościach innych i każda próbuje samodzielnie obsłużyć żądanie. W przypadku, gdy klasa nie jest w stanie w pełni obsłużyć żądania, przekazuje je do następnej i cały proces trwa tak długo, aż komunikat zostanie obsłużony[15]. Podobnie w przypadku omawianego systemu tekst polecenia przekazywany jest najpierw do ogólnego interpretera aplikacji. Tutaj, na podstawie pierwszego wyrazu, rozpoznawany jest rodzaj komendy, następnie pobierany jest interpreter właściwy dla tego rodzaju, po czym komenda pozbawiona zidentyfikowanego słowa przekazywana jest do owego interpretera. Cały proces odbywa się na strukturze przypominającej drzewo. Interpretacja pozwala uszczegółowić rodzaj komendy, po czym na najniższym poziomie następuje pogrupowanie argumentów i przekazanie przetwarzania do właściwego obiektu wykonującego polecenie.

Cały proces tworzenia i pobierania klasy odpowiedniego interpretera opiera się na zawartości statycznego pola z obiektem implementującym wzorzec fabryki. Omawiane pole ma charakter statyczny, ponieważ przechowuje wspólną dla całej klasy informację o interpreterach bardziej szczegółowych, dostępnych dla określonego poziomu rozbioru komendy. Oczywiście klasy znajdujące się na dole hierarchii i zajmujące się tworzeniem obiektów wykonujących polecenia nie mają takiego pola. Od obiektu fabryki przypisanego do statycznego pola interpretera zależy, jakie komendy są obsługiwane przez aplikację. Zamiana domyślnej fabryki na inną oznacza zmianę palety obsługiwanych komend. Rozmiar modyfikacji zbioru poleceń zależy od poziomu, na jakim podmieniany jest obiekt fabryki. W przypadku głównego interpretera programu wymianie podlega cały zestaw poleceń, a w przypadku interpreterów bardziej szczegółowych jedynie pewna rodzina komend.

[15] *Ibidem*, s. 133–134.

## 6. WZORZEC FABRYKI I METODY FABRYCZNEJ UŻYTY W INTERPRETERZE

Wzorzec projektowy fabryki może być implementowany praktycznie w każdym programie napisanym w języku zorientowanym obiektowo. Występuje on w trzech odmianach, a jego podstawową formą jest tzw. fabryka prosta (ang. *simple factory*), której działanie sprowadza się do zwracania instancji klasy odpowiadającej informacjom przekazanym w żądaniu skierowanym do fabryki[16]. Zwracane obiekty wywodzą się od wspólnej klasy bazowej, dzięki czemu jedna i ta sama metoda, po dokonaniu rzutowania w górę hierarchii, może zwracać instancje różnych typów. Zaletą stosowania każdej z odmian fabryki jest uniezależnianie kodu klienta od konkretnych klas, na których wspomniany kod wykonuje operacje. Pociąga to za sobą zwiększenie hermetyzacji obiektów i ułatwia konserwację oraz modyfikację kodu zwracanych instancji. Klient korzystający z dostarczanych przez fabrykę obiektów musi znać jedynie identyfikator potrzebnej mu klasy, na podstawie którego obiekt fabryki zwraca odpowiednią instancję. Dzięki temu ewentualne zmiany w typach już przypisanych do odpowiednich identyfikatorów lub dodanie nowej palety dostępnych klas, oznaczają jedynie modyfikowanie kodu fabryki, podczas gdy korzystający z niej kod klienta może dalej poprawnie działać, bez jakiejkolwiek ingerencji programisty.

Rozwinięciem fabryki prostej jest tzw. fabryka abstrakcyjna (ang. *abstract factory*). Klasy fabryki abstrakcyjnej mają wspólny interfejs i obsługują takie same identyfikatory typów zwracanych instancji. Każda z nich jednak odnosi się do innej grupy klas i w odpowiedzi na ten sam identyfikator może zwracać obiekt klasy o podobnych właściwościach, ale należący do innej grupy. Najczęściej elementem składowym wzorca fabryki abstrakcyjnej jest instancja implementująca fabrykę prostą, która służy do pozyskiwania obiektu konkretnej odmiany fabryki, związanej z pożądaną grupą klas[17]. W opisywanym przykładzie wykorzystuje się przeciążoną funkcję `get` zwracającą instancję rzutowaną do bazowej klasy wszystkich klas w języku Java i przyjmującą jako parametr identyfikatora liczbę całkowitą lub łańcuch znakowy.

Innego interfejsu wymaga trzecia odmiana wzorca projektowego fabryki, określana nazwą metody fabrycznej (ang. *factory method*). W przypadku metody fabrycznej, w odróżnieniu od wcześniej omawianych koncepcji, nie istnieje jedna centralna klasa decyzyjna, określająca jakiego typu powinien być zwracany obiekt. Decyzja o typie dostarczanego obiektu delegowana jest do klas pochodnych, które nie korzystają z identyfikatora, ale zwracają

[16] *Ibidem*, s. 27.
[17] *Ibidem*, s. 41.

instancję klasy odpowiadającą kontekstowi, w którym operują[18]. Dzięki takiemu rozwiązaniu implementacja wzorca metody fabrycznej może być łatwo rozbudowana i dołączana wszędzie tam, gdzie zachodzi potrzeba dostarczenia obiektu o typie odpowiadającym klasom aktualnie wykorzystywanych instancji.

Właściwości metody fabrycznej można wykorzystać przy implementacji wzorca fabryki prostej. Fabryki interpreterów wykorzystywane w omawianej aplikacji neuronowej korzystają z klasy kontenerowej `HashMap`, przechowującej instancje anonimowych klas wewnętrznych, implementujących metodę fabryczną.

Obiekt kontenera typu `HashMap`, jako wspólny dla wszystkich obiektów interpreterów, przechowywany jest w polu statycznym klasy. W sekcji inicjalizacji statycznej do kontenera dodawane są jako klucze identyfikatory klas oraz jako wartości instancje anonimowych klas wewnętrznych, zwracające na żądanie obiekty typu określonego przez identyfikator. Statyczna funkcja `public static void register(String id, FactoryMethod method)` dostarcza prostego sposobu na dodanie nowych instancji klas pochodnych, bazowej klasy metody fabrycznej razem z identyfikatorami. Wspomnianą funkcję można wykorzystać np. w sekcji statycznej klasy dziedziczącej. Metoda `get` dla argumentu całkowitego nie jest obsługiwana i dlatego jej wywołanie powoduje wyrzucenie wyjątku `UnsupportedOperationException()`, oznaczającego w API Javy brak implementacji dla danej operacji.

## 7. WZORZEC POLECENIA JAKO ROZDZIELENIE OPERACJI INTERPRETOWANIA, OBSŁUGI INTERFEJSU I OBLICZEŃ NEURONOWYCH

Interpretowanie polecenia kończy się w omawianym programie stworzeniem instancji klasy obejmującej wszystkie operacje, jakie wiążą się z wprowadzoną komendą. Jest to typowe zadanie dla wzorca projektowego polecenia (ang. *command*), które zamyka w sobie pewien zestaw działań i pozwala na jego wykonanie za pośrednictwem wspólnego, publicznego interfejsu. Zaletą takiego podejścia jest możliwość zgłaszania przez klienta żądań, bez wiedzy na temat operacji jakie zostaną wykonane, a posługiwanie się wspólnym interfejsem pozwala na modyfikowanie zestawu owych operacji bez konieczności ingerowania w kod po stronie klienta[19].

Najlepszym przykładem zastosowania wzorca polecenia jest sterowany zdarzeniami model obsługi graficznego interfejsu użytkownika. W omawianej aplikacji klasy poleceń służą przede wszystkim do logicznego i wygodnego

[18] *Ibidem*, s. 33–34.

[19] *Ibidem*, s. 145–146.

grupowania kodu realizującego różne typy operacji. Konkretne typy poleceń często wymagają podczas procesu konstrukcji podania zestawu argumentów, a cały szereg dodatkowych metod typu `set` ustawia dodatkowe, opcjonalne parametry wykonania komendy. W takiej sytuacji obiekt interpretera musi wiedzieć jakiego typu polecenie powinien stworzyć, dodatkowo sparametryzować i wykonać. Operacja wykonania jest w pewnym sensie przedłużeniem operacji interpretowania, ale rozdzielenie przetwarzania pomiędzy dwie różne klasy ułatwia późniejszą modyfikację i lepiej systematyzuje program.

Obiekt polecenia jest praktycznie jedynym miejscem w programie, gdzie dochodzi do wykonywania operacji wejścia wyjścia. Jedyną interakcją z użytkownikiem, wykraczającą poza klasy poleceń, są komunikaty o błędach, zgłaszane przez interpretery. Komunikaty te są jednak wysyłane na standardowe wyjście błędów, co w przypadku poważniejszej modyfikacji systemu daje względnie duże możliwości przekierowania strumienia. Poważniejsza przebudowa, taka jak np. zmiana interfejsu aplikacji, oznacza głównie wymianę lub modyfikację klas poleceń. Taka architektura systemu wyodrębnia trzy logiczne całości kodu w postaci warstwy klas wykonujących interpretację polecenia, zbioru klas zajmujących się obsługą wykonania komendy pod kątem interakcji z użytkownikiem i systemem plików oraz poziomu właściwych obliczeń neuronowych, których kod jest całkowicie oddzielony od metody sterowania i rodzaju interfejsu aplikacji.

## 8. NEURON JAKO IMPLEMENTACJA WZORCA PROTOTYPU

Sieć neuronowa składa się najczęściej ze sporej liczby zarówno neuronów wykonujących przetwarzanie wstępne oraz końcowe, jak i z właściwych perceptronów. Z tego powodu przekazywanie po jednym obiekcie z każdego typu, wymagane w konstruktorach sieci, nie dostarcza odpowiedniej liczby elementów składowych. Aby mieć możliwość uzyskania dowolnej liczby potrzebnych neuronów, niezbędne jest zaimplementowanie ich w postaci konstrukcyjnego wzorca projektowego prototypu (ang. *prototype*). Wzorzec prototypu pozwala na otrzymanie nowej instancji klasy z obiektu już istniejącego. Proces ten, nazywany klonowaniem, zakłada skopiowanie całości obiektu źródłowego do obiektu docelowego. Klonowanie obiektu, zamiast tradycyjnego tworzenia z użyciem konstruktora, jest szczególnie przydatne, gdy proces konstrukcji wymaga np. dużej ilości czasu i jest dość kłopotliwy, a skopiowanie i odpowiednie zmodyfikowanie zawartości nowego obiektu nie stanowi większego problemu. Przykładem takiej sytuacji może być pozyskiwanie i przetwarzanie odpowiedzi na zapytanie do bazy danych[20].

[20] *Ibidem*, s. 63–64.

Korzystanie z dziedziczenia i właściwości polimorficznych prowadzi też często do operowania lub przechowywania obiektów bez znajomości ich rzeczywistych typów, w postaci rzutowanej do klasy bazowej lub wspólnego interfejsu. Wówczas jedyną możliwością stworzenia dodatkowych obiektów jest właśnie sklonowanie już istniejącego.

W celu zaimplementowania właściwości klonowania w języku Java należy posłużyć się funkcją `protected Object clone()`, należącą do klasy `Object`, będącej typem bazowym wszystkich klas. Ograniczenie zasięgu tej funkcji w postaci identyfikatora `protected` (chroniony) sprawia, że jest ona dostępna wyłącznie dla klas dziedziczących z `Object` i może być wywołana tylko na rzecz tej samej klasy lub własnej klasy bazowej. W celu usunięcia wspomnianego ograniczenia należy w klasie pochodnej przesłonić funkcję `Object clone()`, czyniąc ją jednocześnie metodą publiczną (`public`). W przesłoniętej metodzie możemy następnie wywołać oryginalną funkcję `protected Object clone()` klasy podstawowej. Efektem jej działania jest uzyskanie kopii instancji rzutowanej do podstawowego typu `Object`. Przeprowadzone w ten sposób kopiowanie określa się mianem płytkiego, ponieważ przepisaniu ulega jedynie sam obiekt, podczas gdy zawartość wszystkich pól, z wyjątkiem pól z typami prymitywnymi, wskazuje na te same instancje, do których odwołują się pola obiektu źródłowego. Przeprowadzenie kopiowania głębokiego, czyli przepisanie obiektu razem z całą siecią przypisanych do niego bezpośrednio lub pośrednio instancji, wymaga bardziej złożonej implementacji funkcji `public Object clone()`, dopasowanej ponadto indywidualnie do poszczególnych klas.

Klasa, która może podlegać klonowaniu, musi także implementować interfejs `Cloneable`, który nie pełni w tym przypadku tradycyjnej funkcji interfejsów dostarczających wspólne metody, ale jest swego rodzaju oznaczeniem i potwierdzeniem możliwości klonowania wybranej klasy[21].

W celu dostarczenia możliwości klonowania neuronów w obiektach sieci, w klasie bazowej każdego neuronu uwzględniono wszystkie wyżej wymienione warunki. Oznacza to, że klasa `Neuron` implementuje interfejs `Cloneable`, udostępnia metodę `public Object clone()` i w razie potrzeby wykonuje w przesłoniętej funkcji głębokie kopiowanie obiektu.

## 9. PODSUMOWANIE

Głównym celem podczas implementowania systemu sieci neuronowych i algorytmów ich uczenia było dążenie do osiągnięcia jak największej uniwersalności zarówno pod względem rozbudowy oraz możliwości wielo-

[21] B. Eckel, *Thinking...*, s. 739–745.

krotnego wykorzystania kodu, jak i sposobów używania gotowej aplikacji. Względy wynikające z efektywności kodowania skłoniły do sięgnięcia po język programowania o charakterze obiektowym oraz po koncepcję wzorców projektowych, będącą bardziej skonkretyzowanym rozwinięciem idei obiektowości. Użycie Javy otworzyło ponadto interesujące możliwości rozbudowy aplikacji w kierunku systemów sieciowych oraz rozproszonych, pozwalając jednocześnie na wygodne, wysokopoziomowe oprogramowanie istoty obliczeń neuronowych.

Omawiana aplikacja o roboczej nazwie NeuralEngine została pomyślana jako swego rodzaju silnik neuronowy, możliwy do wykorzystania samodzielnego, w postaci systemu z interfejsem konsolowym, lub jako program współpracujący z innymi modułami. Jej zaletą jest możliwość odczytywania i wykonywania całej sekwencji komend zapisanych w pliku tekstowym, możliwość przetwarzania poleceń dołączonych do jej wywołania w postaci parametrów oraz całkowite sparametryzowanie zaimplementowanych operacji. Sposób obsługi aplikacji można względnie prosto poszerzyć o interfejs graficzny, który może być formą nakładki na już istniejący program bądź wersją rozbudowaną, ingerującą w jego kod. Ingerencja w algorytmy systemu oznacza skorzystanie z implementacji standardowych wzorców projektowych, a konserwacja i rozwijanie programu zbudowanego z takich uniwersalnych i łatwo rozpoznawalnych elementów składowych również nie jest zadaniem skomplikowanym.

Kod programu jest podzielony na trzy logiczne części. Pierwsza z nich, zawarta w pakiecie `rosczak.neuralengine10` obejmuje implementację wzorca projektowego interpretera, służącego do analizy komend wprowadzanych przez użytkownika. Jego struktura jest najbardziej rozbudowana i korzysta w realizowanej strategii z wzorców trzech rodzajów fabryk oraz drzewiastej architektury łańcucha odpowiedzialności. Po zidentyfikowaniu polecenia i jego parametrów, sterowanie przekazywane jest do drugiej części programu, zajmującej się jego wykonaniem oraz interakcją z użytkownikiem i systemem plików. Cały proces odbywa się w obiektach implementujących wzorzec projektowy polecenia i korzystających ze strumieni wejściowych oraz wyjściowych, których operacje zostały rozszerzone na podstawie specjalnych klas dekoratorów. Wykonywanie poleceń wiąże się jednak przede wszystkim z tworzeniem i operowaniem klasami sieci neuronowych, zamkniętymi w oddzielnym pakiecie o nazwie `rosczak.neural10`. Wspomniany pakiet stanowi trzecią logiczną część programu zajmującą się wyłącznie szczegółami obliczeń neuronowych. W celu oddzielenia klas sieci od różnych typów neuronów ich kompozycja odwołuje się do koncepcji wzorca strukturalnego mostu, a konkretne klasy obydwu hierarchii zbudowane są na szkieletach implementujących najczęściej stosowany wzorzec szablonu.

Zastosowane w programie koncepcje oraz mechanizmy mogą oznaczać podniesienie ogólnego poziomu komplikacji i początkowego nakładu pracy, w miarę jednak postępów programowanie zaczyna przypominać składanie całości programu z uniwersalnych elementów, przejrzyście oddających istotę algorytmu i zostawiających cały wachlarz możliwości dalszego rozwoju.

Ostateczną weryfikacją działania i efektywności proponowanego rozwiązania programistycznego jest wykorzystanie opisanego systemu oraz jego elementów składowych przy stworzeniu neuronowej wersji istniejącego modelu ekonometrycznego gospodarki szwedzkiej o nazwie EMIL[22]. Wspomniany model został opracowany przez Prof. Jana B. Gajdę z Uniwersytetu Łódzkiego oraz Prof. Claes-Håkana Gustafsona z Uniwersytetu Örebro, a jego wersję neuronową w postaci programu symulacyjnego w formie apletu Javy można znaleźć w Internecie pod adresem http://pawelrosczak.republika.pl/emil/.

*Paweł Rościczak*

**IMPLEMENTATION OF ARTIFICIAL NEURAL NETWORK ENGINE USING DESIGN PATTERNS IN JAVA**

**(Summary)**

Article presents method of implementation of artificial neural network system by taking advantage of modern method of object oriented programming, and idea of uniform design patterns. Proposed application architecture focuses on providing maximum universality, maximum possibility of further developing and making created neural engine possible to work with other programs.

[22] Więcej na temat modelu EMIL zob. J. B. Gajda, C. H. Gustafson, *EMIL – An Econometric Macro Model of Sweden*, Department of Economics, Örebro University, Working paper No 7, 1999 oraz J. B. Gajda, C. H. Gustafson, *The behaviour of the economy of Sweden from model point of view*, [w:] J. B. Gajda, A. Welfe (eds), *MACROMODELS'97*, Absolwent Printing House, Łódź 1998.