Pisanie kodu dla innych

psychologia poznania, a podstawy pisania dobrego oprogramowania

Piotr Szczepański

na podstawie pracy Thomas Mullen ”Writing Code for Other People Cognitive Psychology and the Fundamentals of Good Software Design Principles”

O czym będziemy mówić

- Wstęp
- Psychologia
  - Kognitywny model
  - Rodzaje pamięci
- Kod
  - Kognitywne strategie
  - Analogie
  - Refaktoryzacje
- Podsumowanie

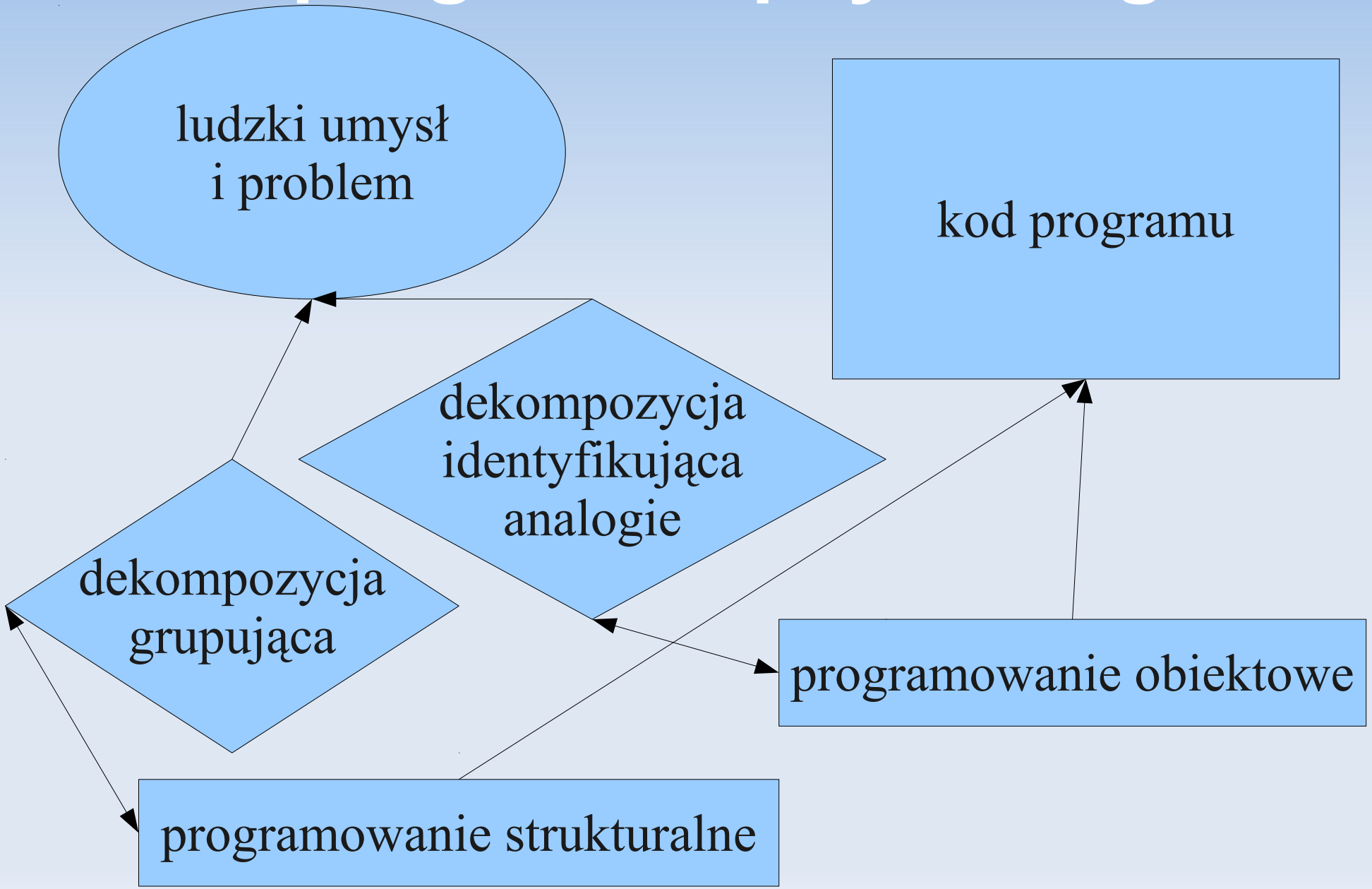
Wstęp

- Ewolucja języków programowania
- Rozwój pryncypiów projektowania oprogramowania

- Odkrywanie sekretów pracy ludzkiego mózgu podczas uczenia się

- Kod jako tekstowa reprezentacja struktur ludzkiej pamięci

Analogie
kod programu - psychologia



Motywacja

- Większość czasu poświęca się na zrozumienie i rozwijanie kodu
  - Pisanie aplikacji zajmuje dużo czasu
  - Rośnie wielkość zespołów programistów
- Powszechna niemożliwość kontaktu z autorem kodu
  - Wymiana kadr w zespole projektowym
  - Rozproszone prace nad projektem
- Wykrywanie błędów

Psychologia kognitywna

- **chunks (klocki)** – grupa elementów pamięci silnie ze sobą powiązanych, a słabo z innymi elementami z pozostałych grup
- **STM** – pamięć krótkotrwała, pracująca; tworzy grupy elementów; zapisuje w LTM
  - limit pojemności
  - limit czasowy (10s)
- **LTM** – pamięć długotrwała; przechowuje wspomnienia
  - Limit zapisywania (2s-8s)

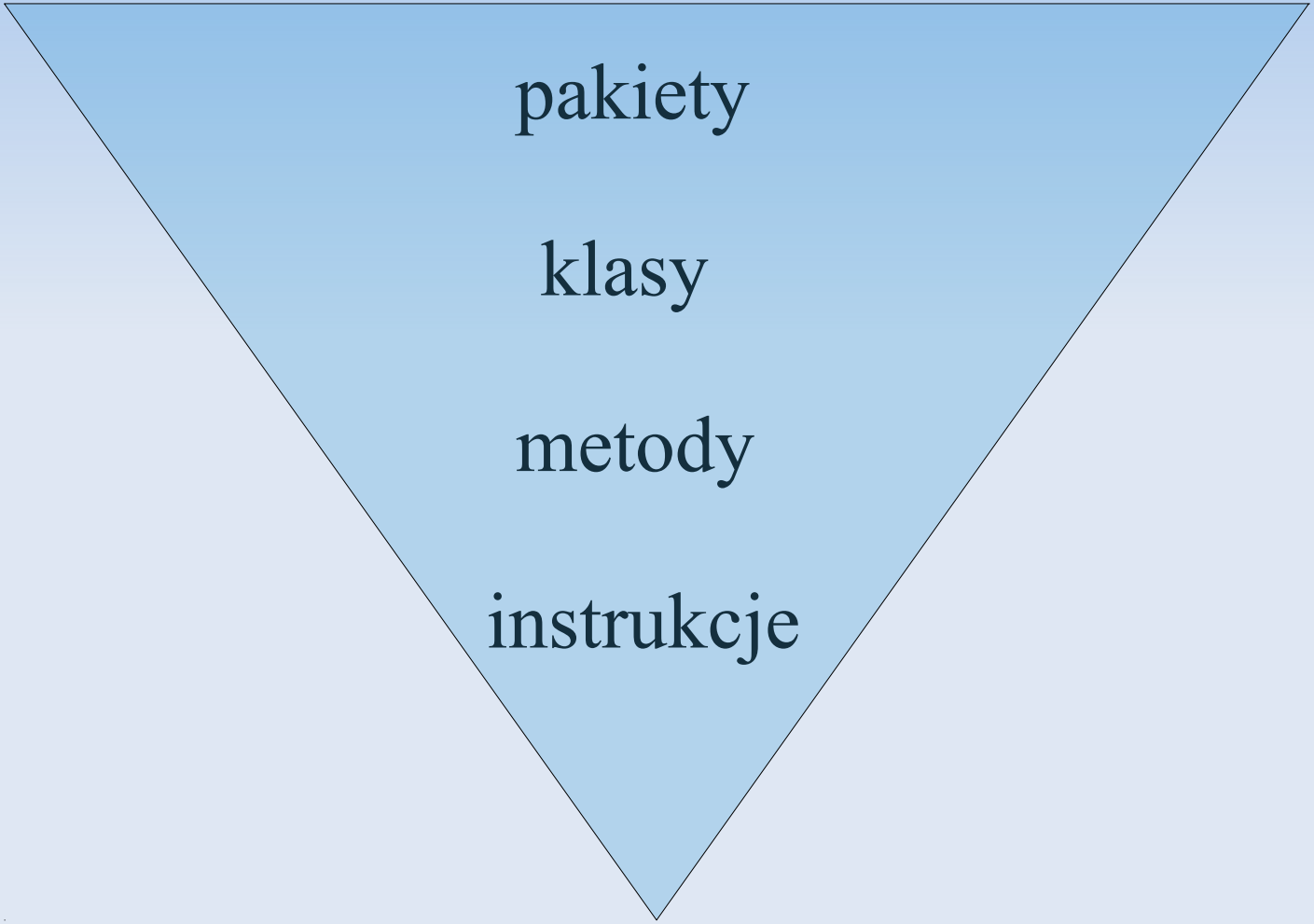
Psychologia kognitywna

- **wiedza ekspercka** – różnice pomiędzy ekspertami, a nowicjuszami w podejściu do rozwiązywania problemu
- **analogiczne rozumowanie** – wszechobecne w ludzkim rozumowaniu; analogie dobierane na podstawie podobieństw struktury i celu

Chunks, a pamięć

- Grupy elementów powinny składać się z 4-5 obiektów
  - Przeciążenie pamięci STM powoduje zamęt
  - Limit ten powoduje optymalną konstrukcję pamięci długotrwałej

Chunks, a Programowanie Obiektowe



Przykładowe grupowanie instrukcji

- Grupowanie po bliskości

```
printHead();
printMsg();


processNew();
update();
```

- Grupowanie po kształcie

```
sum=0;
sumOfSquares=0;
for ( int I=0; I<num; I++)
{
    sum += x[I];
    sumOfSquares += x[I] * x[I];
}
```

Struktura LTM

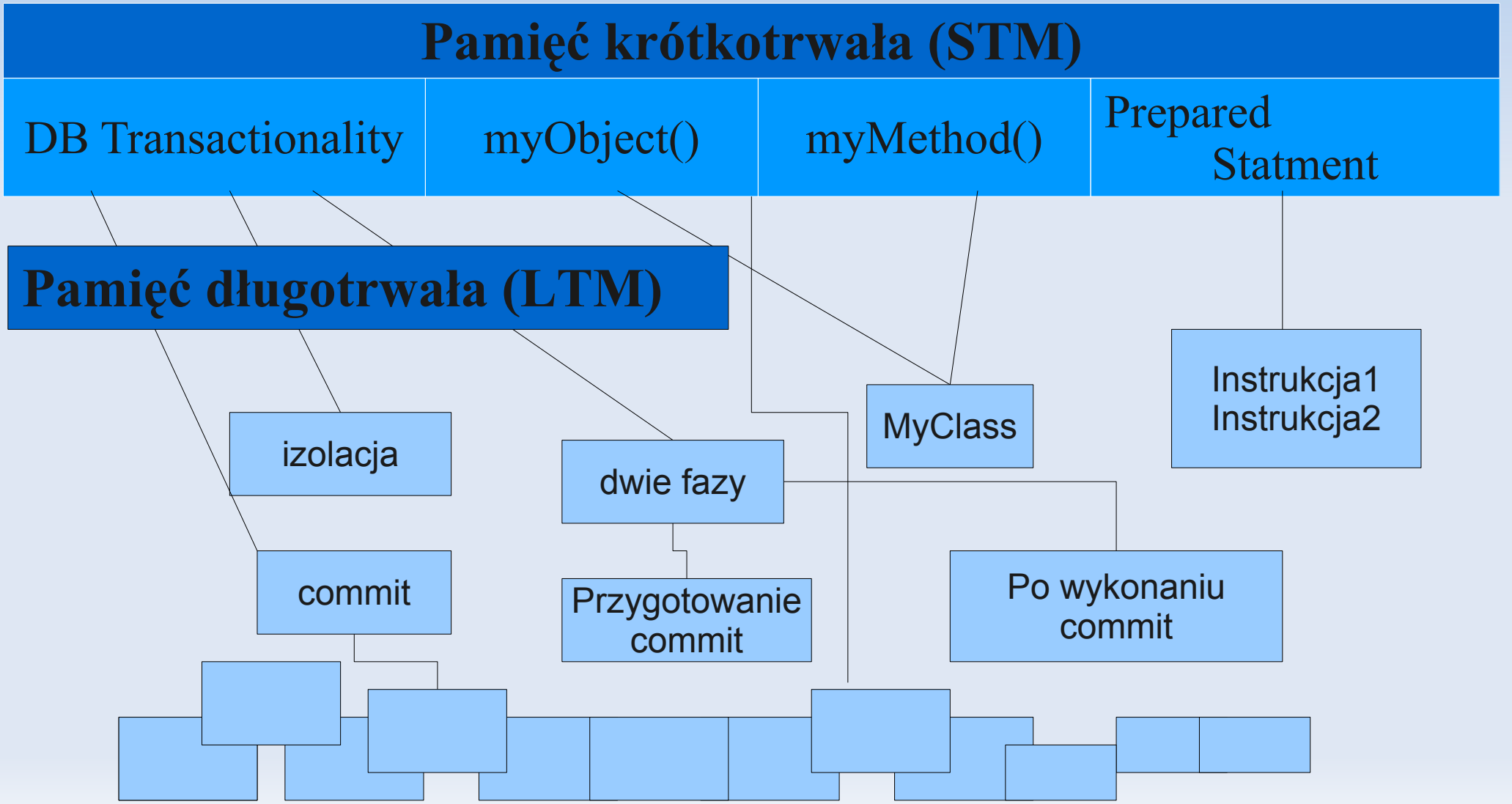
- Sieć wyróżnionych węzłów
  - Podejmowanie decyzji
  - Formułowanie koncepcji
  - Proces rozpoznawania
- Każdy węzeł powiązany z jakąś ideą (litera, słowo, dźwięk, obrazek, odczucie, itp.)
- Poszerzanie sieci, poprzez dodawanie bądź modyfikację węzłów
  - Koszt samej modyfikacji (podstawowy, kilka sekund)
  - Koszt naprawy sieci (powtarzanie)

Co chcemy osiagnać?

- Minimalizacja kosztu naprawy (ponownego uczenia się) sieci
  - Pamięć otwarta na poszerzanie sieci, zamknięta na modyfikację
- Struktura kodu ma naśladować LTM
  - Wzorce projektowe to symptomy pracy ludzkiego mózgu, a nie wytwór oparty na matematycznych algorytmach

Prosty kognitywny model

```
String sql = myObject.myMethod();
PreparedStatement localVar = new PreparedStatement(sql);
localVar.execute();
```



Ewolucja języków

- Języki i paradygmaty programowania prowadzą do powstawania kodu, który jest tekstową reprezentacją rozwiązania problemu, zapisanego w ludzkiej pamięci

Kognitywne strategie

- ”Nadmiarowa informacja”
  - Unikać dających się wyeliminować dodatkowych informacji
  - Zapycha pamięć STM
  - Przykładowo: komentarz do prostej metody getter
- ”Efekt podziału uwagi”
  - Unikać niepotrzebnego rozdzielenia powiązanych rzeczy
  - Pamięć STM zapychana utrzymywaniem połączenia
  - Przykładowo: opis obrazka oddalony od niego samego
  - Metody pośrednie
    - Ułatwia generowanie grup
    - Koszt skakania

Kod dla eksperta i nowicjusza

- Nowicjusz
  - Jak najwięcej kodu pośredniego, grupującego elementy
  - Mało kodu do jednoczesnego oglądania
- Ekspert
  - Umiarkowana ilość kodu pośredniego
  - Dużo kodu do oglądania naraz

Rozumowanie analogiami

- Serce ludzkiej inteligencji
- Analogie opierają się na podobieństwach struktury i celu
- Typy skojarzeń
  - dosłowne podobieństwo
    - Egzemplarze klasy
  - analogie waniliowe (vanilla)
    - Klasy abstrakcyjne
  - analogie zasad
    - Set jako zbiór dowolnych elementów

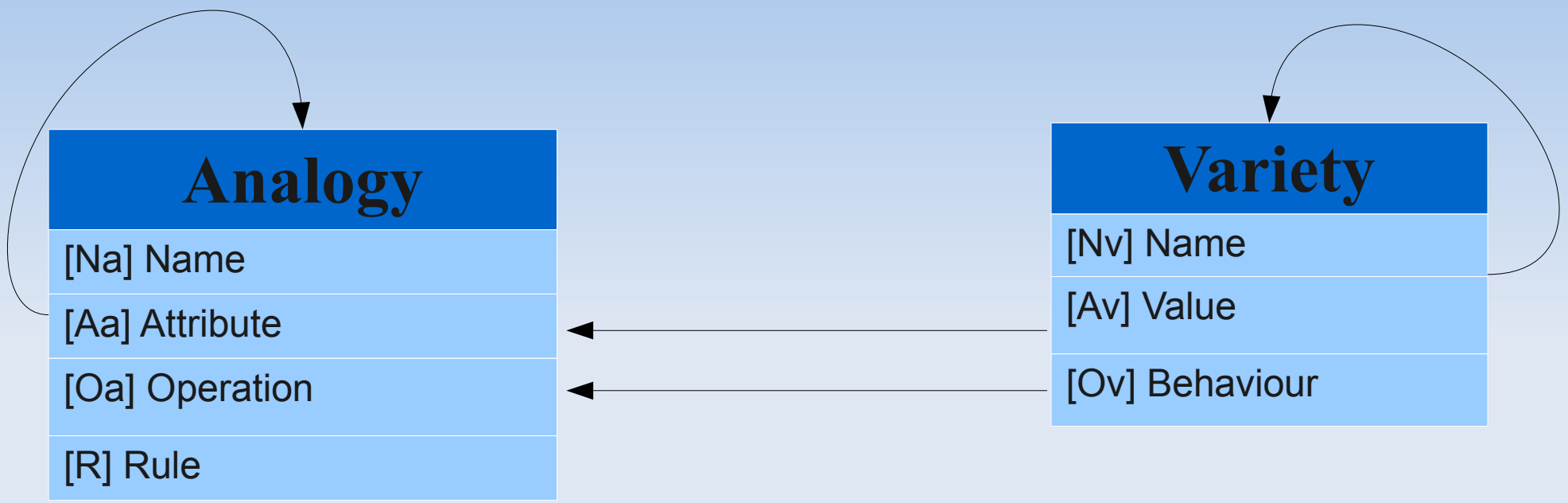
Wykrywanie analogii

- Dobrym przewodnikiem w identyfikacji analogii w programie jest możliwość prostego rozszerzenia go, właśnie poprzez analogię.

```
Public String getAccountType(String account)
{
   if ( exchange.equals(”EUREX”) )
      return getEurexAccountType(account);
   if ( exchange.equals(”LIFFE”) )
      return getLiffeAccountType(account);
}
```

  - Poszerzyć o wymianę ”Matif”

Analogie


Analogy
[Na] Name
[Aa] Attribute
[Oa] Operation
[R] Rule
Variety
[Nv] Name
[Av] Value
[Ov] Behaviour

Analogia „wyłącznego atrybutu”

```
Static Map exchangeCountry = new HashMap();

static {
    exchangeCountry.put(”EUREX”,”Germany”);
    . . .
}

public String logMessage (String exchangeName)  {
    return ”Exchange ” + ” exchangeName + ” is in ” +
        exchangeCountry.get(exchangeName);
}
```

- Nazwa analogii – *exchange*
- Atrybut – *country*
- Nazwa zmiennej – np: ”EUREX”
- Wartość – np: ”Germany”

Analogia „kształtu wyrażenia”

```
Private boolean isValid() {
    if  (expirationDate == null)
        return false;

    if (contactNumber == null)
        return false;
...
}
```

- Nazwa analogii – n/a
- Operacja – isValid()
- Nazwa zmiennej – nazwa pola do sprawdzenia
- Zachowanie – instrukcje sprawdzające pole

Analogia „przełącznika”

```
public double calculate (double first, double second) {
    switch (operand) {
        case MULTIPLY:
            return first * second;
        case DIVIDE:
            return first / second;
        ...
    }
    return 0;
}
```

- Nazwa analogii – n/a
- Operacja – calculate()
- Nazwa zmiennej – np: DIVIDE
- Zachowanie – blok związany z case

Analogia „nazwy metody”

- Prefiks:

```
public void visitExpression(Node a){};
public void visitBlock(Node a){};
public void visitFile(Node a){};
```

- Sufiks:

```
double a = doubleObj.doubleValue();
int b = doubleObj.intValue();
long c = doubleObj.longValue();
```

- Nazwa analogii – n/a
- Operacja – np: ”visit”
- Nazwa zmiennej – np: ”Expression”
- Zachowanie – blok metody

Analogia „argumentu metody”

```
Float f = Math.max(1.0F, 2.0F);
Int i = Math.max(1, 2);
Long l = Math.max(1L, 2L);
```

Operacja – nazwa metody (max)

Nazwa zmiennej – typ parametru

Zachowanie – implementacja metody

Analogia „klasy/interfejsu”

- Nazwa analogii – nazwa klasy
- Atrybut – pola klasy
- Operacja – metody abstrakcyjne
- Nazwa zmiennej – nazwy klas dziedziczących
- Wartość – wartość pól
- Zachowanie – implementacje abstrakcyjnych metod

Inne analogie

- analogia „reguły”
  - Typy generyczne w Javie (np. Kolekcje)
- analogia „aplikacji”
  - Podobieństwo pomiędzy uruchomionymi egzemplarzami programu
  - Atrybuty np. pliki konfiguracyjne

Jak oceniać jakość kodu

- Zasada czterech elementów
  - Oszczędza czytelnikowi tworzenia własnych grup elementów
- Dobrze zastosowane analogie
  - Stwarzanie elementów wyższej abstrakcji

Refaktoryzacje

- **chunking**
  - Nie więcej niż 4 elementy w grupie
- **nazewnictwo zmiennych i metod**
  - W sieci LTM nazwa będzie używana jako idea powiązana z węzłem
- **nazwanie grupy**
  - Identyfikuje nieoczywistą grupę elementów
- **redukcja niepotrzebnych nazw pośrednich**
  - Oszczędność pamięci STM

Refaktoryzacja

- **extract method**
  - Tworzy i nazywa grupę elementów
- **inline method**
  - Redukuje niepotrzebną informację (nazwę)
- **inline temp**
  - Pozbycie się niepotrzebnych zmiennych pomocniczych
- **substitute algorithm**
  - Wychwytywanie analogii

Podsumowanie

- Analogie pomiędzy kognitywnym modelem poznania, a projektowaniem wzorców
- Odnajdowanie analogii oraz grupowanie elementów
- Przestrzeganie zasad, ułatwia tworzenie kodu prostego do zrozumienia
- Czy kod programu jest dobrym odwzorowaniem kognitywnego modelu, czy może ewolucja kodu programów uformowała współczesną postać tego schematu ludzkiego poznania?