1. Znaczące nazwy

a. używaj nazw przedstawiających intencje

Powinna informować, w jakim celu istnieje, co robi i jak jest używana.
Jeżeli nazwa wymaga komentarza, to znaczy, że nie ilustruje intencji.

b. unikaj dezinformacji

Powinniśmy unikać słów, które wprowadzają znaczenia różniące się od oczekiwanego.

Np. nie należy nazywać grupy kont mianem *accountList*, o ile nie jest to faktycznie zmienna typu *List.*
Dlatego lepiej użyć po prostu *accounts*.

Należy unikać nazw, które nieznacznie się od siebie różnią.

Przedstawianie podobnych koncepcji za pomocą podobnych sekwencji znaków to informacja.

c. tworzenie wyraźnych różnic

Jeżeli utworzymy klasę, o nazwie *Product* i *ProductData* lub *ProductInfo*, otrzymamy różne nazwy,
ale nie spowodujemy, że będą one znaczyły coś innego.

Dodatkowe słowa są nadmiarowe. Słowo *variable* nigdy nie powinno pojawiać się w nazwie zmiennej, a *table* w nazwie tabeli.

d. nazwy łatwe do wyszukania

Bardzo łatwo można wyszukać ciąg znaków *MAX_CLASSES_PER_STUDENT*, ale liczba 7 może być problematyczna.
Bardzo ciężko znaleźć też jednoliterowe. Takie powinno się używać tylko jako lokalne.

e. nazwy klas

Klasy i obiekty powinny być rzeczownikami lub wyrażeniami rzeczownikowymi (NIE czasownikami).
Należy unikać słów takich jak: *Manager*, *Processor*, *Data*, *Info*.

f. nazwy metod

Metody należy opatrywać nazwami będącymi czasownikami lub wyrażeniami czasownikowymi.

Gdy konstruktory są przeciążone, należy używać metod fabryk o nazwach opisujących argumenty.
Na przykład:

```
Complex fulcrumPoint = Complex.FromRealNumber(23.0);
```

jest zwykle lepsze od:

```
Complex fulcrumPoint = new Complex(23.0);
```

Warto rozważyć wymuszenie stosowania tej techniki przez zadeklarowanie odpowiedniego konstruktora jako prywatnego.

g. jedno słowo na pojęcie

Należy stosować zasadę jedno słowo na jedno abstrakcyjne pojęcie i trzymać się jej.
Spójny leksykon jest ogromnym ułatwieniem dla programistów, którzy muszą korzystać z naszego kodu.

Na przykład mylące jest stosowanie nazw *fetch*, *retrieve* i *get* do analogicznych metod w różnych klasach.

h. nie twórz kalamburów

Należy unikać używania tego samego słowa do dwóch celów.

i. korzystanie nazw z dziedziny rozwiązania

Należy korzystać z terminów informatycznych (nazw algorytmów, wzorców, terminów matematycznych itd.).
Niekoniecznie trzeba używać każdej nazwy z dziedziny problemu, aby współpracownicy nie musieli często pytać klienta o znaczenie.

Nazwa *AccountVisitor* niesie ze sobą wiele informacji dla programisty, który zna wzorzec ***visitor***. Nazwa *JobQueue* już niekoniecznie.

j. korzystanie nazw z dziedziny problemu

Wszędzie tam, gdzie nie istnieją terminy programistyczne dla wykonywanych operacji, należy używać nazw z dziedziny problemu.

2. Funkcje (1/2)

a. małe funkcje

Tak małe, aby były przejrzyste i oczywiste. Każda opowiada pojedynczą historię.
Każda prowadzi do następnej we wzorowym porządku.

b. jedna czynność (single responsibility SRP)

Funkcje powinny wykonać JEDNĄ operację. Powinny robić tylko to.

Jeżeli funkcja wykonuje tylko operacje znajdujące się o jeden poziom poniżej zadeklarowanej nazwy funkcji, to wykonuje ona jedną operację:

```
public static String renderPageWithSetupsAndTeardowns(PageData pageData, boolean isSuite) {
  if (isTestPage(pageData)) {
    includeSetupAndTeardownPages(pageData, isSuite);
  }
  return pageData.getHtml();
}
```

c. jeden poziom abstrakcji

Mieszanie poziomów abstrakcji w jednej funkcji zawsze jest mylące.
Mogą być problemy z rozpoznaniem, czy określone wyrażenie jest ważnym zagadnieniem, czy mało istotnym szczegółem.

d. zasada zstępująca

Po każdej funkcji znajduje się kolejna, na następnym poziomie abstrakcji, dzięki czemu można czytać program,
schodząc o jeden poziom abstrakcji niżej wraz z przejściem do kolejnej funkcji w pliku.

```
foo1() { foo21(); foo22(); }
foo 21() { foo31(); foo32(); }
foo22() { foo31(); }
foo31() { ...
```

e. instrukcje switch

ZŁY przykład:

```
switch (e.type) {
  case COMMISSIONED: return calculateCommissionedPay(e);
  case HOURLY:       return calculateHourlyPay(e);
  default:           throw new InvalidEmployeeType(e.type);
}
```

Wady:
- szansa, że inne funkcje również potrzebują takie samej struktury,
- wykonuje więcej niż jedną operacja (złamanie zsady pojedynczej odpowiedzialności SRP),
- trzeba zmienić przy dodaniu nowego typu (złamanie zasady otwarty-zamknięty OCP),
- Instrukcja *switch* powoduje, że metoda rośnie.

Rozwiązaniem tego problemu jest ukrycie instrukcji switch w podstawie *fabryki abstrakcyjnej*.
Fabryka użyje instrukcji *switch* do tworzenia odpowiednich obiektów typów bazujących na interfejsie *Employee*,
a różne jej funkcje będą rozmieszczone polimorficznie przez interfejs *Employee*.

2. Funkcje (2/2)

f. argumenty funkcji

Idealną liczbą argumentów dla funkcji jest zero. Następnie jeden i dwa.
Należy unikać konstruowania funkcji o trzech argumentach.
Więcej niż trzy argumenty wymagają specjalnego uzasadnienia, a i tak nie powinny być stosowane.

Im więcej argumentów, tym trudniejsze do zrozumienia i tym większe problemy z ich kolejnością.

Gdy funkcja wymaga więcej niż dwóch argumentów, prawdopodobnie niektóre z nich powinny być umieszczone w osobnej klasie.

Argumenty znajdują się na innym poziomie abstrakcji niż funkcje i wymusza to zapoznanie się ze szczegółami.

Argumenty są kłopotliwe z punktu widzenia testowania. Trudno napisać wszystkie testy jednostkowe zapewniające,
że wszystkie kombinacje argumentów będą działały prawidłowo.

Argumenty wyjściowe są trudniejsze do zrozumienia niż argumenty wejściowe.
Zwykle zakłada się, że dane wchodzą do funkcji przez argumenty i wychodzą z funkcji poprzez zwracaną wartość.

g. argumenty znacznikowe (boolean)

Przekazanie wartości boolean do funkcji od razu komplikuje sygnaturę metody, informując,
że funkcja wykonuje więcej niż jedną operację zależnie od wartości *true*/*false*.

h. argumenty wyjściowe

Argumenty są w naturalny sposób interpretowane jako dane wejściowe do funkcji.

Jeżeli funkcja musi zmieniać stan czegokolwiek, powinna zmieniać stan własnego obiektu.

i. unikanie efektów ubocznych

Nie powinno mieć miejsca, że funkcja obiecuje, że wykonuje jedną operację, ale oprócz tego realizuje inną w sposób ukryty.

Czasami w niespodziewany sposób modyfikuje zmienną ze swojej klasy.
Czasami zmienia parametry przekazane do funkcji lub globalne zmienne systemowe.

j. rozdzielanie poleceń i zapytań

Funkcje nie powinny jednocześnie coś wykonywać i odpowiadać na jakieś pytanie
Funkcja powinna zmieniać stan obiektu ALBO zwracać pewne informacje na temat tego obiektu.

k. wyodrębnienie bloków try-catch

Bloki try-catch naruszają strukturę kodu i mieszają przetwarzanie błędów ze zwykłym przetwarzaniem.
Z tego powodu warto wyodrębniać treść bloków try i catch do osobnych funkcji.

l. obsługa wyjątków jest jedną operacją

Obsługa błędów jest jedną operacją, dlatego funkcja obsługi błędów nie powinna wykonywać nic innego.

Jeżeli w funkcji istnieje słowo kluczowe *try*, powinno być pierwszym słowem w funkcji
i nie powinno się w niej znajdować nic poza blokami *catch* i *finally*.

m. nie powtarzaj się

Programowanie strukturalne, programowanie aspektowe, programowanie komponentowe
— są po części strategiami eliminowania powtórzeń.

3. Obiekty i struktury danych

a. abstrakcja danych

Ukrywanie implementacji polega na tworzeniu abstrakcji.

Ukrywanie implementacji nie sprowadza się do dodawania warstwy funkcji nad zmiennymi.
Klasa nie powinna po prostu przepychać zmiennych przez gettery i settery.
Zamiast tego powinna udostępniać interfejs pozwalający użytkownikom na manipulowanie istotą danych bez konieczności znajomości jej implementacji.

b. asymetria danych i obiektów

Obiekty ukrywają dane, tworząc abstrakcje, i udostępniają funkcje operujące na tych danych.
Struktury danych udostępniają ich dane i nie mają znaczących funkcji.

Kod obiektowy ułatwia dodawanie nowych klas bez zmiany istniejących funkcji;
trudnia dodawanie nowych funkcji, ponieważ muszą zostać zmienione wszystkie klasy.
Kod proceduralny (kod korzystający ze struktur danych)
ułatwia dodawanie nowych funkcji bez zmiany istniejących struktur danych;
utrudnia dodawanie nowych struktur danych, ponieważ muszą zostać zmienione wszystkie funkcje.

Kod strukturalny łamie zasadę open/close (OCP), dlatego powinien być stosowany w wyjątkowych przypadkach,
gdy mamy pewność, że nie nie będzie dodawana nowa struktura danych.

c. prawo Demeter

Prawo Demeter głosi, że metoda **f** klasy **C** powinna wywoływać tylko metody z:

- **C**,
- obiektu utworzonego przez **f**,
- obiektu przekazanego jako argument do **f**,
- obiektu umieszczonego w zmiennej instancyjnej klasy **C**.

```
class C {
  private MyClass _instance;

  public void f(MyClass  arg) {
    MyClass local;
    // …
    _instance.foo();                    // OK, bo _instance jest obiektem zmiennej instancyjnej, czyli jeden z 4 ww. przypadków
    _instance.getPerson().getName();   // złamanie prawa Demeter ! jeśli MyClass i Person nie są trukturami
  }
  private MyClass innerFunction() { … }
}
```

Moduł powinien nic nie wiedzieć o wnętrzu obiektów, którymi manipuluje.
Oznacza to, że obiekt nie powinien udostępniać swojej struktury wewnętrznej przy użyciu akcesoriów,
ponieważ powoduje to udostępnienie, a nie ukrycie wewnętrznej struktury.

Metoda nie powinna wywoływać metod z obiektów zwracanych przez jakąkolwiek z innych dozwolonych funkcji.

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

To, czy naruszone jest prawo Demeter, zależy od tego, czy ctxt, Options i ScratchDir są obiektami,czy strukturami danych.
Jeżeli są **obiektami**, to wiedza na temat jej budowy wewnętrznej jest naruszeniem prawa Demeter.
Jeżeli są po prostu **strukturami danych** bez zdefiniowanych operacji, to prawo Demeter nie ma tu zastosowania.

d. DTO - Data transfer Object

Klasa ze zmiennymi publicznymi niezawierająca funkcji. DTO są przydatnymi strukturami, szczególnie w przypadku komunikowania się z bazami danych, przesyłania danych między warstwami (zwłaszcza przez Internet) itp.

4. Klasy

a. organizacja klas, hermetyzacja

1. lista zmiennych
    1.1. statyczne publiczne
    1.2. statyczne prywatne
    1.3. instancyjne prywatne
    1.4.publiczne (stosuje się bardzo rzadko)
2. metody publiczne
3. funkcje prywatne

b. hermetyzacja

Zazwyczaj zmienne i funkcje użytkowe pozostają prywatne.
Czasami jednak zachodzi konieczność zmiany na chronioną, aby była dostępna dla testu. Testy powinny ustalać zasady.

Na początku jednak zawsze należy szukać sposobu na zachowanie prywatności.
Rozluźnianie zasady prywatności jest ostatnim możliwym rozwiązaniem.

c. małe klasy (single responsibility SRP)

Nazwa klasy powinna opisywać pełnione przez nią odpowiedzialności.

Nazewnictwo jest prawdopodobnie pierwszym sposobem na pomoc w określeniu wielkości klasy.
Jeżeli nie możemy utworzyć spójnej nazwy klasy, prawdopodobnie jest ona zbyt duża.
Im bardziej ogólna jest nazwa klasy, tym większe prawdopodobieństwo, że ma wiele odpowiedzialności.

d. spójność

Klasy powinny mieć niewielką liczbę zmiennych instancyjnych.
Każda z metod klasy powinna manipulować jedną lub kilkoma tymi zmiennymi.
Zwykle im większą liczbą zmiennych manipuluje klasa, tym bardziej spójna z klasą jest ta metoda.

Zwykle nie jest zalecane ani możliwe tworzenie takich maksymalnie spójnych klas; z drugiej strony, spójność powinna być wysoka.
Gdy spójność jest wysoka, oznacza to, że metody i zmienne klasy są wzajemnie zależne i tworzą logiczną całość.

5. Obsługa błędów

a. rozpoczynanie od try-catch-finally

Spróbujmy napisać testy wymuszające wyjątki, a następnie dodać do procedury obsługi operacje pozwalające spełnić te testy.
Spowoduje to konieczność zbudowania zakresu transakcji za pomocą bloku *try*,
co pomoże nam utrzymać transakcyjną naturę tego zakresu.

b. niekontrolowane wyjątki

Ceną kontrolowanych wyjątków jest naruszenie zasady open-close.
Jeżeli zgłosimy kontrolowany wyjątek w metodzie, a instrukcja *catch* znajduje się kilka poziomów wyżej,
musimy zadeklarować ten wyjątek w sygnaturze każdej metody pomiędzy naszym kodem a catch.
Oznacza to, że zmiana wykonana na niskim poziomie oprogramowania wymusza zmiany na wielu wyższych poziomach.

c. dodawanie kontekstu za pomocą wyjątków

Trzeba tworzyć komunikaty błędów zawierające odpowiednie informacje i przekazywać je za pomocą wyjątków. .
Stack trace nie zawiera informacji o przeznaczeniu nieudanej operacji, dlatego należy przekazać wystarczające informacje,
aby można było zarejestrować błąd w bloku catch.

d. definiowanie normalnego przebiegu - wzorzec Special Case Pattern

Tworzymy w nim klasę lub konfigurujemy obiekt, aby obsługiwał szczególny przypadek za nas.
Po takiej operacji, nie trzeba obsługiwać sytuacji wyjątkowej.
Obsługa ta jest hermetyzowana w obiekcie przypadku specjalnego.

e. nie zwracamy null

Gdy zwracany jest ***null***, w wymagany jest dodatkowy nakład pracy, aby go obsłużyć w funkcjach wywołujących.
W takich przypadkach brak jednego testu wartości null powoduje, że aplikacja wymyka się spod kontroli.

Jeżeli mamy zamiar zwrócić *null* z metody, warto rozważyć zgłoszenie wyjątku lub zwrócenie ***obiektu specjalnego przypadku***

Przykład:

```
public List<Employee> getEmployees() {
  if( .. brak pracowników .. )
    return Collections.emptyList();
  }
}
```

f. nie przekazujemy null

Należy wystrzegać się przekazywania null.

*O ile nie korzystamy z API, które oczekuje wartości *null*.