ZAŁĄCZNIK: EITE_SMILE_Standardy_wytwarzania_oprogramowania_EBOK

Cel dokumentu

Dokument zawiera spis reguł, praktyk i standardów stosowanych w trakcie wytwarzania oprogramowania dla systemu EBOK/PS opartego na Oracle Application Development Framework.

Zastosowane skróty i pojęcia

Skrót / Pojęcie	Objaśnienie
CC&B	Oracle Customer Care and Billing solution
WCP	Web Center Portal
EOB	Energa Obrót S.A.
EOP	Energa Operator S.A.

Zarządzanie kodem

Na potrzeby zarządzania kodem Wykonawca i Dostawca otrzyma dostęp do repozytorium kodów źródłowych. Wykonawca i Dostawca będzie zobowiązany wszystkie zmiany nanosić w udostępnionym repozytorium, wykorzystując osobne gałęzie kodu dla każdej spójnej zmiany.
Wykonawca i Dostawca będzie zobowiązany oznaczać w repozytorium wersje oprogramowania gotowe do weryfikacji przez EITE oraz wdrożenia na środowiska Energa.

Dokumentacja zmian

Zmiany wprowadzane w kodzie źródłowym będą wynikać bezpośrednio z Projektu Technicznego lub zgłoszenia serwisowego do Wykonawcy lub Dostawcy. Wykonawca i Dostawca będą mieli jednak możliwość wprowadzenia zmian w kodzie źródłowym z ich inicjatywy, w momencie wykrycia w kodzie źródłowym błędów lub po prostu w celu poprawy jego jakości (refaktoryzacja, optymalizacja, itp.). W przypadku Wykonawcy wymaga to jednakże zgłoszenia Zamawiającemu planowanej zmiany przed przygotowaniem paczki z oprogramowaniem. Wykonawca i Dostawca jest zobowiązany oznaczać w kodzie źródłowym zmiany nanoszone w ramach zgłoszeń serwisowych przynajmniej numerem zgłoszenia oraz krótkim uzasadnieniem wykonywanej zmiany. Oznaczeniem takim może być komentarz – zawierający numer zgłoszenia i uzasadnienie – do rewizji w repozytorium kodu, w ramach której zmiana została wprowadzona.

Komentowanie kodu

Wykonawca i Dostawca jest zobowiązany do komentowania kodu. Komentarze maja w sposób nie budzący wątpliwości opisywać działanie modułów, funkcji, procedur i przekształceń. Treść komentarzy powinna być zapisana w języku polskim lub angielskim.

Weryfikacja kodu

EITE będzie mogło przeprowadzać weryfikację dostarczonego kodu źródłowego pod względem zgodności ze standardami kodu źródłowego, bezpieczeństwa, wydajności, zgodności z dobrymi praktykami programowania oraz poprawnym jego udokumentowaniu.

Standardy

Ogólne zasady Kodowania Java

Zasady ogólne

Pliki .java wykorzystywane w projekcie EBOK umieszczane są w pakiecie zaczynającym się od pl.energa.portal

Pliki .java wykorzystywane w projekcie PS umieszczane są w pakiecie zaczynającym się od pl.energa.sos.ps

Językiem dla kodu jest język angielski. Wszystkie nazwy klas/interfejsów, metod, zmiennych, treści komentarzy powinny być w tym języku. Dopuszcza się odstępstwo od tych reguł w uzasadnionych przypadkach (np. cytowane nazwy plików dokumentacji, słowa nie mające jasnego tłumaczenia, np. PESEL). Treść komentarzy powinna być zapisana w języku polskim.

W plikach używamy znaków ASCII lub UTF-8, znakiem końca linii jest znak zgodny z Windows.

Organizacja plików

Pliki źródłowe

Każdy plik źródłowy zawiera jedną publiczną klasę, interfejs lub enumerację. Jeżeli prywatne klasy, interfejsy lub enumeracje są powiązane z klasą publiczną to umieszczamy je w pliku z klasą publiczną. Klasa publiczna powinna być pierwszą klasą (lub interfejsem) w pliku.

Każdy plik źródłowy składa się z następujących sekcji:

- Nazwa pakietu i definicje import
- Deklaracja klasy lub interfejsu

Sekcje powinny być rozdzielone pustą linią

Nazwa pakietu i definicje import

Sekcja zawiera klauzulę package a po niej następuje sekcja klauzul import. Klauzule import mogą nie wystąpić.

package pl.energa.sos.ps.ejb.jmsqueue;

import pl.energa.sos.ps.utilsProject.JaxbHelper;

import pl.energa.sos.ps.utilsProject.JmsQueue;

Deklaracja klasy lub interfejsu

Poniższa tabela przedstawia części składowe definicji klasy lub interfejsu, które powinny występować w przedstawionej kolejności

Deklaracja klasy/interfejsu	Komentarz
Klauzula class lub interface lub enum	

Komentarz implementacyjny (/*...*/), jeśli istnieje taka konieczność	Komentarz zawiera informacje dotyczące całości klasy, które nie nadają się do komentarza dokumentacyjnego.
Deklaracje zmiennych statycznych (static)	W kolejności zależnej od poziomu dostępu: public, protected, package, private.
Deklaracje zmiennych instancji klasy	W kolejności: public, protected, package, private.
Konstruktory klasy	
Metody klasy	Metody powinny być pogrupowane po funkcjonalności a nie w zależności od poziomu dostępności. Metody prywatne (usługowe) powinny być zdefiniowane po pierwszej metodzie (ale niekoniecznie zaraz po), która je używa.

Wcięcia

Minimalną jednostką wcięcia są 4 spacje.

Nie używamy tabulatora do definicji wcięcia.

Linię zaczynającą nowy blok przesuwamy o jedną jednostkę wcięcia w prawo.

Kolejną linię kodu umieszczamy na tym samym poziomie co początek poprzedniej klauzuli, jeśli nie zaczynamy nowego bloku kodu.

Długość pojedynczej linii

Linia nie powinna być dłuższa niż 160 znaków.

Zawijanie wierszy

Jeżeli wyrażenie nie mieści się w pojedynczej linii, należy je umieścić w wielu liniach kierując się następującymi zasadami:

- Łamiemy po przecinku
- Łamiemy przed operatorem
- Załamaną część wiersza zaczynamy od miejsca, w którym zaczyna się łamane wyrażenie lub lista atrybutów (patrz przykład)
- Jeżeli kod po złamaniu przesuwa się za bardzo do prawego marginesu, można zrezygnować z wyrównania do początku łamanego wyrażenia lub listy atrybutów i zastosować wcięcie minimum 8 spacji
- Wcięcie po złamaniu ma co najmniej 8 spacji.
- Kolejne łamanie linii dla zagnieżdżonego wyrażenia, dla którego nie było jeszcze łamania powoduje kolejne wcięcie. Nowa linia zaczyna się w miejscu, gdzie jest początek łamanego wyrażenia.
- Staramy się łamać linię na poziomie wyrażenia mniej zagnieżdżonego, niż bardziej zagnieżdżonego

Przykłady:

Łamanie wyrażenia będącego parametrami wywołania metody:

```
someMethod(someParameter1, someParameter2,
        someParameter3, someParameter4,
        someParameter5); // wielokrotne łamanie tego samego wyrażenia na tym samym poziomie zagnieżdżenia nie powoduje kolejnego wcięcia

var = someMethod1(someParameter1,
```

```
            someMethod2(someParameter2,
                    someParameter3));    // łamanie zagnieżdżonego wyrażenia daje kolejne wcięcie

public static synchronized someGreatAndLongMethodNameMethod(Object parameter1,
                Object parameter2,
                Object parameter3,
                Object parameter14) {          // złamane linie zaczynamy wcześniej niż '(', bo przybliża nam kod do
prawego marginesu

Łamanie wyrażenia arytmetycznego:

someVariable1 = someVariable2
            / (someVariable3 - someVariable4 - someVariable5)
            + 7 * someVariable6;                          // Łamanie przed operatorem

someVariable1 = someVariable2 / (someVariable3 - someVariable4
                - someVariable5) + 7 * someVariable6;        // Źle, można było złamać na wyższym poziomie
wyrażenia jak wyżej

    ...
}
```

Komentowanie kodu

Stosujemy poniższe sposoby komentowania kodu:

- komentarze kodu blokowe /* ... */
- komentarze kodu końca linii // ...

Komentarze implementacyjne używamy do opisania zastosowanych rozwiązań implementacyjnych, dla których sam kod nie stanowi wystarczającej dokumentacji.

Komentarz implementacyjne używamy również do logicznego podział kodu na sekcje. Komentarz ma umożliwić zorientowanie się, jaki krok procesu wykonujemy kodem następującym po komentarzu.

Komentarzy nie stosujemy do upiększenia kodu (gwiazdki, szlaczki, ramki, itp.), tylko do przekazania treści nie wynikających w oczywisty sposób z kodu.

Forma komentarzy implementacyjnych

Komentarz blokowy

Stosujemy go przeważnie do opisu całej klasy/interfejsu/metody zaraz po ich deklaracji oraz do dokumentacji całego pliku na jego początku.

Komentarz blokowy zawiera na początku jedną linię bez tekstu, aby był czytelnie oddzielony od poprzedzającego go kodu.

```
/*
 * Komentarz zaczyna się w drugiej linii, aby pierwsza była wizualnym
 * rozdzielnikiem kodu.
 */
```

Komentarz jednoliniowy

Wcięcie komentarza jednoliniowego powinno być takie jak kod, który po nim następuje.

Jeśli komentarza nie da się napisać w jednej linii to powinno stosować się komentarz blokowy.

Komentarz jednoliniowy może być komentarzem blokowym lub komentarzem końca linii

```
if (someCondition) {
    /* do something for someCondition */
    ...
}
albo
if (someCondition) {
    // do something for someCondition
    ...
}
```

Komentarze wtrącane

Krótkie komentarze mogą wystąpić w linii kodu, której dotyczą. Taki komentarz musi być wyraźnie odsunięty od kodu, który opisuje.

Jeżeli występuje kilka komentarzy tego typu i dotyczą tego samego kawałka kodu, to komentarze powinny zaczynać się w tej samej kolumnie.

Komentarz może być komentarzem blokowym albo komentarzem końca linii.

```
if (a == 2) {
    return TRUE;            /* all is done already, can return */
} else {
    return isMyNumber(a);     /* check other conditions */
}
albo
if (a == 2) {
    return TRUE;            // all is done already, can return
} else {
    return isMyNumber(a);     // check other conditions
}
```

Komentarz końca linii

Komentarz stosujemy do wykomentowania całego bloku kodu, którego nie chcemy usuwać.

Komentarz stosujemy do oddzielania sekcji logicznych kodu.

Komentarz stosujemy jako komentarz jednoliniowy albo wtrącany (jak opisano w poprzednich sekcjach)

...

```
//Generate car accident
// first check some conditions
if (someCondition) {
```

```
    // Do something for that condition
    ...
}
else {
    return true;          // all was already done
}
//if (someCondition) {
//
//    // Do something unpredictable
//    ...
//}
//else {
//    return false;
//}

// go on road and crash
goCrash();

// leave the car if You can
if (badEnjured()) {
    ...
}
```

Deklaracje

Jedna deklaracja na linię

Preferowanym stylem deklaracji zmiennych, jest jedna w jednej linii (umożliwia komentowanie):

```
Object someObj;      // object for ...
Object secondOne;    // placeholder for...
zamiast
Object someObj, secondOne;
```

Niedopuszczalna jest definicja zmiennych różnych typów w jednej linii:

```
    Object someObj, secondOne[];
```

Inicjalizacja

Staramy się zainicjalizować zmienną w miejscu gdzie ją tworzymy (oczywiście, jeśli się da).

Umieszczenie deklaracji

W blokach kodu metod umieszczamy deklarację w miejscu pierwszego użycia. Wyjątek stanowi deklaracja zmiennej w ciele pętli - robimy ją poza ciałem.

Unikamy deklaracji przesłaniającej zmienną zdefiniowaną w bloku wyżej.

```
someMethod() {
    int count = 10;
    ...
```

```
    if (condition) {
        int count = 0;     // nie dopuszczamy do takiej deklaracji
        ...
    }
    ...
}
```

Deklaracje klas i interfejsów

Przyjmujemy następujące zasady:

- Nie ma przerw między nazwą a nawiasem "(" zaczynającym parametry.
- Nawias rozpoczynający blok kodu "{" jest umieszczony na końcu linii, dla której definiuje blok kodu.
- Nawias zamykający blok kodu "}" pisany jest w nowej linii z takim samym wcięciem jak klauzula, dla której zamyka blok kodu.

```
class SomeClass extends Object {
    int var1;
    int var2;

    Sample(int i, int j) {
        var1 = i;
        var2 = j;
    }

    int someEmptyMethod() {            // uwaga, puste bloki kodu też podlegają tej zasadzie
    }

    ...
}
```

Wyrażenia

Wyrażenia proste

Jedna linia powinna zawierać jedno wyrażenie

```
argv++;          // Prawidłowo
argc--;          // Prawidłowo
argv++; argc--;   // ŹLE
```

Wyrażenia złożone

Wyrażenie złożone to blok kodu otoczony nawiasami { }. Poniższy przykład pokazuje sposób formatowania bloku kodu dla różnych konstrukcji java (takich jak if, while, do, for), w następującym zakresie:

- pozycja znaku '{' - na końcu linii, w której występuje wyrażenie, dla którego blok jest definiowany
- pozycja znaku '}' - w kolejnej linii z wcięciem takim samym jak wyrażenie, dla którego blok jest definiowany
- wcięcie dla wyrażeń w bloku - jeden poziom więcej
- blok definiowany zawsze - nawet jeśli składa się z jednego lub żadnego wyrażenia w bloku
- spacja oddziela słowa kluczowe od nawiasów
- nawiasy '{' i '}' oddzielone są spacją od reszty kodu

```
if (condition) {          // spacje między 'if' a '('
    doSomething();
}

while (doSomething()) {   // pusty blok
}

if (condition) {
    doSomething();        // pojedyncze wyrażenie też jako blok
} else {
    doSomething();
    doMore();
}

do {
    doSOmething();
} while (someCondition);

if (condition) {
    doSomething;
} else if (condition) {   // kolejny przykład na użycie spacji i nawiasów różnego typu
    doSomething();
} else {
    doSomething();
}
```

return

return nie powinien używać nawiasów, chyba że chcemy podkreślić zwracaną wartość w nieoczywistym przypadku jakim jest zastosowanie konstrukcji ? :

```
return myResultList.size();

return (size ? size : defaultSize);
```

Zamiast wyrażenia:

```
if (booleanExpression) {
    return true;
} else {
    return false;
}
```

stosujemy:

```
return booleanExpression;
```

Podobnie zamiast:

```
if (condition) {
    return x;
}
return y;
```

stosujemy:

```
return (condition ? x : y);
```

switch

Poprawna forma przedstawiona jest poniżej. Należy zwrócić uwagę na następujące aspekty:

- wcięcie 'case' w porównaniu do 'switch'
- wcięcie dla wyrażeń wykonywanych dla odpowiednich case
- umiejscowienie 'break'
- wymagany komentarz, jeżeli intencjonalnie chcemy przejść do kolejnego 'case'
- słowo 'default' - występuje zawsze, nawet jeśli nic nie wykonujemy w tym bloku.
- break; występuje nawet przy ostatniej opcji 'switch' - w poniższym przykładzie jest to opcja 'default'
- brak nawiasów '{' i '}' dla kodu wykonywanego w poszczególnych opcjach ('case', 'default')

```
switch (condition) {
   case ABC:
      doSomethingForABC();
      /* falls through */

   case DEF:
      doSomething();
      doSomethingMore();
      break;

   case XYZ:
      doSomethingXYZ();
      break;

   default:
      doSomethingByDefault();
      break;
}
```

try-catch

```
try {
   statements;
} catch (ExceptionClass e) {
   statements;
} finally {                    // użycie finally jest opcjonalne
   statements;
}
```

Białe znaki

puste linie

Puste linie stosujemy jako rozdzielniki podnoszące czytelność kodu.

Używamy dwóch lub jednej linii odstępu w zależności od wpływu na czytelność.

Stosujemy puste linie aby:

- Oddzielić od siebie sekcje pliku źródłowego
- Oddzielić od siebie kolejne metody
- Zaznaczyć logiczne części w kodzie metody

- Pogrupować deklaracje zmiennych
- Oddzielić komentarz od poprzedzającego kodu

spacje

Spacji używamy w następujących przypadkach:

- Oddzielamy nawiasy od słów kluczowych
- Stosujemy do zaznaczenia wcięć
- Stosujemy po przecinku w liście argumentów
- Oddzielamy operatory binarne (oprócz kropki oraz) od elementów, na których wykonują operacje.
- Oddzielamy wyrażenia w definicji dla 'for'
- Oddzielamy deklarację rzutowania od rzutowanego obiektu

```
while (true) {
   a += c + d;
   a = (a + b) / (c * d);

   for (int d = 0; d < 100; d++) {
      n++;
   }
   printSomething("Some text " + n + "\n", a);
   callAnother((byte) aNumber, (Object) obj);
}
```

Gdzie NIE używamy spacji:

- Nie oddzielamy nawiasu '(' od nazwy metody, którą definiujemy lub wołamy
- Nie oddzielamy nawiasu '(' od zawartości, która po nim następuje
- Nie stosujemy spacji przed znakiem nawiasu ')'

Konwencja nazewnicza

Typ obiektu	Reguły nazewnictwa	Przykłady
Pakiet	zaczyna się od pl.energa.portal lub pl.energa.sos.ps. Stosujemy małe litery, chyba, że nazwa pakietu zawiera kilka słów. Używamy wtedy wielkiej litery dla każdego słowa w nazwie, które nie jest pierwszym.	pl.energa.sos.ps.ejb.jmsqueue
Klasa/Interfejs	Nazwa może składać się z wielu słów. Każde słowo piszemy wielką literą, bez rozdzielania znakiem '_', pozostałe litery małe. używamy dla przyrostków '_Impl', '_Test', '_Gen' mających narzucone znaczenie przez środowisko	OutcomingServicesTemplate
Metoda	Nazwa może składać się z wielu słów. Każde słowo, oprócz pierwszego, piszemy wielką literą, bez rozdzielania znakiem '_', pozostałe litery małe.	validateSoftParameters(); run();
Zmienna	Format nazwy podobnie jak nazwa metod. Nazwa powinna nieść informację, do czego zmienna jest używana, albo jakie dane przechowuje. Dla nieznaczących zmiennych tymczasowych (np. licznik pętli) można używać nazw jednoliterowych	int i; CharacteristicType_Id charTypeId;
Stała	Używamy tylko wielkich liter. Kolejne słowa oddzielamy znakiem '_'. Każda stała jest statyczna i finalna	static final int MIN_WIDTH = 40;

Dodatkowe zasady

Nie robimy publicznych zmiennych w klasie, bez wyraźnego powodu. Operacje na zmiennych powinny być wykonywane poprzez metody a nie bezpośrednio na zmiennych. Wyjątek może stanowić klasa, którą definiujemy w celu stworzenia struktury danych i która nie niesie ze sobą żadnych operacji.

Do metod i zmiennych statycznych klasy odwołujemy się poprzez definicję klasy a nie jej instancję.

Stałe (tzw. literały) nie powinny być używane bezpośrednio w kodzie. Zasada nie dotyczy -1, 0, 1, które mogą występować w pętlach jako inicjatory licznika.

Nie przypisujemy wartości do wielu zmiennych na raz. (np. myVar = loopcopunter = MyClass.MAX_LOOP_COUNTER;)

Nie stosujemy operatora przypisania tam gdzie łatwo może być pomylony z operatorem porównania. (np. zamiast if (myVar = outVar) { napisać if ((myVar = outVar) != 0) {)

Jeżeli przed operatorem '?' mamy wyrażenie wymagające wyliczeń to wyrażenie to umieszczamy w nawiasach (np. (x >= 0) ? x : -x;)

KONWENCJA NAZEWNICTWA OBIEKTÓW W BAZIE DANYCH

Tabela

SS_NAZWA lub SP_NAZWA

np:

- SS_CONFIGURATION (tabela konfiguracyjna)
- SP_PROCESS (tabela z procesami sprzedawców)

Dodatkowo dla każdego projektu istnieją tabele o nazwach:

DATABASECHANGELOG

DATABASECHANGELOGLOCK

NAZWA_KOLEJKI_WLSTORE

np:

SP_JMS_WLSTORE

Kolumny w tabeli

Każda tabela ma klucz główny o nazwie:

- Nazwa tabeli bez prefiksu (SS lub PS) _ID np. tabela SS_ORDER -> kolumna ORDER_ID

Jeżeli tworzymy nowe pole to:

- wszystkie znaki są UPPERCASE
- wyrazy oddzielamy znakiem "_" (tz. podkreślenie/"kładka")

Klucze PK/FK

Tworząc nowe więzy integralności kierujemy się następującą konwencją:
<NAZWA_TABELI>_PK (w przypadku klucza głównego)
Np. SS_LINK_PK
<NAZWA_TABELI1>_<NAZWA_TABELI2>_FK_<n> (w przypadku klucza obcego)
,gdzie:
NAZWA_TABELI1 – nazwa tabeli w której ten klucz obcy jest założony,
NAZWA_TABELI2 – skrócona nazwa tabeli, na którą klucz obcy wskazuje
n - kolejny numer constraint'a

np. SS_LINK_SS_USER_DATA_CHANG_FK1

Sekwencje

Nazwa sekwencji powinna zawierać
- Nazwę tabeli do której będzie wykorzystywana
- Sufiks (ID_SEQ)
- Przykład:

Np. SS_USER_DETAIL_ID_SEQ dla tabeli SS_USER_DETAIL

Indeksy

Nazwa indeksu powinna zawierać
- Skróconą nazwę tabeli
- Skróconą nazwę pola zawierającą pierwsze litery każdego wyrazu w polu
- Sufiks (IDX)
- Przykład:

Np. SS_DIC_LIN_DHI_IDX dla tabeli SS_DICTIONARY_LINE I kolumny DICT_HEADER_ID

KONWENCJA W SKRYPTACH BAZODANOWYCH

Plik wejściowy

Główny plik wejściowy programu liquibase powinien nazywać się:
<NAZWA_PROJEKTU><RODZAJ_SCHEMATU>_run.xml
Gdzie
<NAZWA_PROJEKTU> - sp_eop lub eob
<RODZAJ_SCHEMATU> - app lub adm
Plik powinien mieć rozszerzenie xml i zawierać:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<databaseChangeLog >
        <include file="[nazwa_folderu]/[nazwa_pliku]" />
        <include file="[nazwa_folderu]/[nazwa_pliku]" />
</databaseChangeLog>
```

, gdzie
[nazwa_folderu] - nazwa folderu, w których znajdują się pliki. Nazwa ta powinna wyglądać w sposób następujący:
changes_<NAZWA_PROJEKTU><RODZAJ_SCHEMATU> np. changes_sp_eop_app
[nazwa_pliku] - nazwa kolejnych plików tworzonych podczas wydawania paczki

Pliki ze skryptami

Każdy plik tworzony podczas wydawania paczki powinien się nazywać zgodnie z następującą konwencją:

AA.BB.CC.xml
Gdzie
AA – NUMER WYDANIA
BB – NUMER KOLEJNEJ PACZKI W TYM WYDANIU
CC – NUMER FIXA DO OBECNEJ PACZKI.
Np. Dla wydania 3, paczki 2 i 3 fixa plik powinien mieć nazwę:
03.02.02.xml
Każdy plik powinien zawierać:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<databaseChangeLog>
        <changeSet author="[nazwa_autora]" id=" [numer_changeseta]">
                [nazwa_funkcji]
        </changeSet>

        <changeSet author="[nazwa_autora]" id=" [numer_changeseta]">
                <tagDatabase tag="[numer_tagu]"/>
        </changeSet>
</databaseChangeLog>
```

, gdzie
<changeSet/> może wystąpić jeden lub więcej razy w pliku
[nazwa_autora] – nazwa osoby piszącej tę funkcję
[nazwa_funkcji] – nazwa wykonywanego zapytania SQL
[numer_changeseta] – kolejny numer funkcji wg. Poniższej konwencji:
AA.BB.CC.<NUMER_ZMIANY>
, gdzie NUMER_ZMIANY to trzy-cyfrowy numer zaczynający się od 1.
[numer_tagu]" – AA.BB.CC
W każdym pliku ostatnim changeSetem musi być changeSet z wartością <tagDatabase>

Pomiędzy kolejnymi changeSetami powinna być linia przerwy. Każdy changeSet powinien być wcięty w stosunku do tagu <databaseChangeLog>
Nazwa funkcji powinna być wcięta w stosunku do tagu <changeSet>

KONWENCJA aplikacji adf/ webcenter portal

Wstęp
Każda aplikacja adf/wcp powinna się składać z następujących projektów:
- Projektu z Modelem ADF zawierającym Klasy z encjami, Message Driven Bean
- Projektu ViewController zawierającym taskflowy, pliki z widokami, pliki javascript, pliki css oraz ManagedBeany
- Projektu Utils zawierającym wszystkie klasy pomocnicze
- Projektu WS, który wystawia WS
- Projektu z Clientem WS

Przyjęte konwencje nazewnictwa są zgodnie z dokumentem:
http://www.oracle.com/technetwork/developer-tools/adf/learnmore/adf-naming-layout-guidelines-v2-00-1904828.pdf

Model ADF
- **Nazwy obiektów encyjnych** – powinny pasować do obiektu bazodanowego, ale zamiast znaku podkreślenia powinien być zastosowany system notacji camelCase rozpoczynający się od wielkiej litery. Na przykład tabela SS_LINK powinna być odwzorowywana na klasę SsLink.
- **Atrybuty obiektów encyjnych** – nazwy atrybutów powinny odpowiadać kolumnom w bazie danych, ale zamiast znaku podkreślenia powinien być zastosowany system notacji camelCase np. Zamiast CAST_TYPE powinna być zmienna castType.

- **Generator Sekwencji** – powinien mieć następującą nazwę <NAZWA_SEKWENCJI>_GEN np:
Dla sekwencji „SP_USER_SEQ" powinien być generator „SP_USER_SEQ_GEN"

ViewController

- Nazwy managed bean – nazwa managed bean powinna odpowiadać nazwie widoku lub task flowu, którego dotyczy. Nie powinno się w nazwie beana umieszczać jego zasięgu. Ponadto nazwa managed bean powinna zawierać sufiks MB np. widokowi archive.jspx powinien odpowiadać managed bean ArchiveMB.java
- **Nazwa managed bean w taskflow** – nazwa managed bean w taskflow powinna być identyczna jak nazwa klasy z tym wyjątkiem, że pierwsza litera powinna być mała.
- Nazwa page definition – nazwa page definition dla widoku powinna być identyczna jak nazwa widoku, lecz dodatkowo powinna zawierać sufiks PageDef i mieć rozszerzenie .xml
- Wszystkie pliki css powinny mieć rozszerzenie .css
- Wszystkie pliki javascript powinny mieć roszerzenie .js
- **Wszystkie pliki z tłumaczeniami** tekstów powinny mieć sufiks [nazwa_jezyka].properties
Gdzie [nazwa_jezyka] to kod języka (dla Polski pl)
- **nazwy widoków** – wszystkie nazwy widoków powinny mieć rozszerzenie .jspx, .jsp lub .jsff (dla fragmentów)
- unbounded task flow powinien zawsze się nazywać adfc-config.xml i powinien być tylko jeden w aplikacji