ZAŁĄCZNIK: EITE_SMILE_Standardy_wytwarzania_oprogramowania_CCB

Cel dokumentu

Dokument zawiera spis reguł, praktyk i standardów stosowanych w trakcie wytwarzania oprogramowania dla systemu Oracle Customer Care & Billing opartego na Oracle Utilities Application Framework.

1 Zastosowane skróty i pojęcia

Skrót / Pojęcie	Objaśnienie
CC&B	Oracle Customer Care and Billing solution
Dostawca	Zewnętrzny dostawca oprogramowania w zakresie systemu SMILE.
EOB	Energa Obrót S.A.
EOP	Energa Operator S.A.

2 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.

3 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.

4 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.

5 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.

6 Standardy

6.1 Ogólne zasady Kodowania Java

6.1.1 Zasady ogólne

Pliki java rozszerzające funkcjonalność CC&B umieszczane są w pakiecie zaczynającym się od com.splwg.cm.

Niniejszy standard nie dotyczy plików i treści generowanych automatycznie (np. klas generowanych interfejsów czy treści Adnotacji, @BatchJob, itp.).

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.

6.1.2 Organizacja plików

Pliki źródłowe

Każdy plik źródłowy zawiera jedną publiczną klasę lub interfejs. Jeżeli prywatne klasy lub interfejsy 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:

- Komentarz nagłówkowy
- Nazwa pakietu i definicje import
- Deklaracja klasy lub interfejsu

Sekcje powinny być rozdzielone pustą linią

Komentarz nagłówkowy

Wszystkie pliki źródłowe powinny zaczynać się od komentarza (c-style), który ma poniższą zawartość:

```
/*
 * Confidentiality Information
 *
 * PROGRAM DESCRIPTION
 *
 * CHANGE HISTORY
 *
 * Date:       by:       Reason:
 * YYYY.MM.DD  login     Name of document the creation/modification is based on.
 */
```

Nazwa pakietu i definicje import

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

```
package com.splwg.cm.domain.billing.batch;

import java.math.BigInteger;
import java.util.List;
```

Deklaracja klasy lub interfejsu

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

Deklaracja klasy/interfejsu	Komentarz
Komentarz dokumentacyjny (javadoc /**...*/)	Szczegóły w części 'Komentarz dokumentacyjny'
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.

6.1.3 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

   ...
}
```

6.1.4 Komentowanie kodu

Stosujemy wszystkie możliwe sposoby komentowania kodu:

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

Dla każdej klasy/interfejsu/enumeracji oraz każdej metody piszemy komentarz javadoc, który potem będzie widoczny w Przeglądarce Aplikacji uruchamianej z poziomu CC&B.

Komentarze dokumentacyjne używamy do krótkiego opisania dostarczanej funkcjonalności klasy/interfejsu/metody oraz parametrów, wyjątków i zwracanego wyniku. W komentarzu dokumentacyjnym należy umieścić również informacje o ograniczeniach i przyjętych założeniach co do działania kodu, jeśli te informacje nie wynikają z innych elementów systemu.

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()) {
   ...
}
```

Komentarz dokumentacyjny

Format komentarza dokumentacyjnego musi być zgodny z formatem przyjętym dla narzędzia javadoc http://java.sun.com/products/jdk/javadoc/

Wcięcie komentarza dokumentacyjnego ma być takie jak kod, którego komentarz dotyczy.

Dla opisywania metod używamy co najmniej znaczników @param, @return, @throws (o ile występują do tego przesłanki)

6.1.5 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
    }

    ...
}
```

6.1.6 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;
}
```

6.1.7 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 ')'

6.1.8 Konwencja nazewnicza

Typ obiektu	Reguły nazewnictwa	Przykłady
Pakiet	zaczyna się od com.splwg.cm. Stosujemy małe litery, chyba, że nazwa pakietu zawiera kilka słów. Używamy wtedy wielkiej litery dla każdego słowa w nazwie, które nie jest pierwszym.	com.splwg.cm.domain.customBusinessComponent
Klasa/Interfejs	Nazwa może składać się z wielu słów. Każde słowo piszemy wielką literą, bez rozdzielania znakiem '_', pozostałe litery małe. Każda klasa zaczyna się od przedrostka Cm. Znaku '_' używamy dla przyrostków '_Impl', '_Test', '_Gen' mających narzucone znaczenie przez środowisko	CmLatePaymentInterestCalculationBatch, CmFeatureConfigurationRoutine_Impl, CmDataXmlDefinictionMaintenance_Test
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 '_'	static final int MIN_WIDTH = 40;

6.1.9 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;)

6.2 Java Programming Standards

Źródło: JavaProgrammingStandards.doc

Contents

Rationale
Guidelines
Naming Standards

6.2.1 Rationale

In order to make it easier for programmers working on the same codebase to easily read each other's (and their own!) code, we need to enforce certain standard coding conventions. These conventions will also be helpful when comparing code revisions under version control, as the code should be formatted consistently and no irrelevant formatting-related differences will appear in the diff.

6.2.2 Guidelines

First, Sun has their own code standards guidelines here: http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html. Like most coding guidelines, these are quite reasonable and differ only in minor details from other guidelines.

The web page http://geosoft.no/development/javastyle.html also has some very nice tips. Note that we won't prefix instance variable names with underscores--instead we use Eclipse syntax coloring to make ivars easily visible.

We use the prefix *fetch* in method names in entity implementation classes, in order to perform object navigations that aren't already defined by Hibernate mappings.

Here are some additional notes:

Not surprisingly, a lot can be learned from good Smalltalk style. The books "Smalltalk With Style" (Klimas, Skublics, Thomas) and "Smalltalk Best Practices Patterns" (Kent Beck) provide a lot of good ideas for code organization and naming that are applicable to Java as well as Smalltalk.

All code should be:

- Written with tabs equal to 4 spaces, not "hard" tabs. Each level of indentation should be one "tab".
- Generally free of hard-coded "magic" strings or numbers (e.g. max number of items in some list). If you need such a string or number value, you should use (or create) a constant or property.

Classes should use specific, not package-based imports, where practical. I.e. import com.foo.UsefulClass, not com.foo.*.

Variables should generally be private. Only create accessor (e.g. get/set) methods when absolutely needed ("Dont reveal your private parts").

Prefix "getter" methods with "get", e.g. "getFoo()", setters with "set", e.g. "setBar(aBar)". Don't use "Flag" or "Switch", or abbreviations thereof, i.e. "getAllowedSw()" should be "getIsAllowed(), and "setAllowedSw(aBoolean)" should be "setIsAllowed(aBool)".

Use camel-case instance and parameter variable names, without underscore prefixes or suffixes (do use uppercase for constants, as suggested in the guidelines reference above). Instance variables start with lower-case letters.
Methods should generally be public or private (again, to allow future subclassing). Use of interfaces is encouraged to declare useful sets of public methods.
Don't abbreviate except for standard industry abbreviations (i.e. HTML, HTTP). Use long, meaningful class, method, and variable names.
Methods should be short and clear. Instead of placing comments before a section of code in a method, rather create another method that describes what is being done by the method name.
When using Java API collections, reference them through generic interfaces, not specific implementation classes, e.g.

```
List someList = new ArrayList();
...
Map someMap = new HashMap();
...
```

This lets you change your mind about implementation (e.g. ArrayList to LinkedList) without breaking any code.

6.2.3 Naming Standards

Contents

1. General guidelines
2. Entity Naming Guidelines
3. Collection Naming Guidelines
4. Lookup Naming Guidelines
5. Java/COBOL Naming Guidelines
6. Special Cases

6.2.3.1 1. General guidelines

- Don't use reserved java words
- Don't use spaces
- Don't abbreviate
- Don't use punctuation
- Don't start the name with a number

Here are our project guidelines for naming properties:

- Generally, don't abbreviate. The exceptions are SA, SP when the name would get too long if written as e.g. *ServiceAgreement* as part of a much longer field name
- In line with the above, spell out **amount** and **total**
- Boolean values (SW) are prefixed with **is**, **has**, **can**, **are**, or **should**, according to what is grammatically correct.
- Date fields end with **Date**
- Time fields end with **Time**
- Datetime fields end with **DateTime**
- Id is spelled **Id**
- Don't include a final *Flag* (FLG) or *Code* (CD)
- Use **min** instead of minimum, and **max** instead of **maximum**
- Can be generic- that is, for the field BILL_STATUS, you can just name it **status**

6.2.3.2 2. Entity Naming Guidelines

- Be specific- the name MUST be unique
- Language tables (_L) don't need to be named
- Don't append "View" to a view.
- Don't abbreviate

- Don't use plural names (e.g. BillMessages)

6.2.3.3 3. Collection Naming Guidelines

Contents

3.1. Class Name
3.2. Collection Name

6.2.3.3.1 3.1. Class Name

The class name for a collection includes the owning entity name and the collection name in singular form.
<owning_entity><collection_name_in_singular_form>

Examples:

- **AdjustmentTypeAlgorithm**
- **AdjustmentTypeCharacteristic**
- **BillableChargeTemplateLine**

6.2.3.3.2 3.2. Collection Name

For collections, the one-off generation created a large number of collection names. Many of these are overly verbose, and should be shortened. Simply modify the collectionName in the entity annotation. Here are guidelines:

- Shorten **adjustmentTypeAlgorithms** to **algorithms**
- Shorten **adjustmentTypeCharacteristics** to **characteristics** (in rare cases you may have more than one kind of characteristic, in which case you need more specific names)
- Remove the owning entity name from the front of the collection name, e.g. **billableChargeTemplateLines** becomes **lines**

6.2.3.4 4. Lookup Naming Guidelines

Here are guidelines for naming Lookups (on the Lookup Field maintenance):

- Be specific- the name MUST be unique across all lookups
- Don't include a final standard suffix *Flag* or *Lookup* (The suffix Lookup is automatically added by the generator to the classes generated for each Lookup field.)
- Examples:
  - WO_STATUS_FLG -> writeOffStatus
  - STM_RTG_METH_FLG -> statementRoutingMethod

Here are guidelines for naming Lookups Value properties (on Lookup Value maintenance):

- Try to word the name in a way that makes sense when prepended by *is*, and is also valid when standing alone as a constant. (eg {isComplete, COMPLETE}, {isFrozen, FROZEN})
- The name might match the english description of the lookup value.
- Examples:
  - HOW_TO_USE_FLG : - -> subtractive
  - ITEM_STATUS_FLG : A -> active
  - DGRP_PRIO_FLG : 10 -> highest10
  - DGRP_PRIO_FLG : 20 -> priority20

6.2.3.5 5. Java/COBOL Naming Guidelines

When coding Java classes for Java -> Cobol processing, the following standards should be applied.

- The Cobol program Java class should be named "CobolProgramCIPXXXXX_Impl" where CIPXXXXX is the name of the Cobol program being called. It should reside in the package "com.splwg.cis.cobol.XX" where XX is the current subsystem id (i.e. CI, AD, etc.).
- The Cobol copybook Java class should be named "CobolCopybookCICXXXXX_Impl" where CICXXXXX is the name of the Cobol copybook being used. It should reside in the package "com.splwg.cis.cobol.XX" where XX is the current subsystem id (i.e. CI, AD, etc.).

6.2.3.6 6. Special Cases

6.2.3.6.1 6.1. 'Type' Entity Controlling Characteristics for 'Instance' Entities - Characteristic Controls

There are 'type' entities that control the characteristics for their 'instance' entities. These are tables typically named **CI_CHTY_<type_entity>**, e.g., CI_CHTY_CCTY. These type entities specify a list for its instances the valid characteristic types, default characteristic types, required characteristic types, etc. This list is the type entity's **Characteristic Controls**.

The following are the naming conventions for the characteristic controls:

Characteristic control class	**<type_entity>CharacteristicControl**
Characteristic control collection	**characteristicControls**

For example, the class name for characteristic control of Customer Contact Type is **CustomerContactTypeCharacteristicControl**.

And the collection is defined as follows:

```
/**
 * @version $Revision: #1 $
 * @BusinessEntity (tableName = CI_CC_TYPE,
   oneToManyCollections = { @Child (collectionName = characteristicControls, childTableName = CI_CHTY_CCTY)})
*/
```

6.3 HQL Programming Standards

Źródło: HQLProgrammingStandards.doc

The applications use an object relational mapping library called Hibernate (information available at http://www.hibernate.org/). This library handles persistence operations against the database for changed entities, and also provides a querying language.

The Hibernate Query Language (http://www.hibernate.org/hib_docs/reference/en/html/queryhql.html) provides a more object oriented approach to querying against the database. Joins can more clearly be indicated via “navigation” to the related foreign key, letting hibernate fill in the join when it constructs the SQL.

Note that in most situations only a subset of the hibernate query language is used. For instance, when constructing a query whose order is important, the query must programmatically specify the order by, as opposed to placing the order by clause into the HQL itself. This allows the application to perform additional operations upon the HQL that may be required for different databases, and also to apply validations to the HQL.

Here are some examples of creating and using queries. The convenience methods to create the query are available on any “context managed object”- that is, entities, change handlers, business components, maintenances, and the implementer extensions of any of them.

To select all algorithms with a given algorithm type:

```
AlgorithmType algorithmType = … ;
Query query = createQuery(“from Algorithm algorithm where “ +     “algorithm.algorithmType = :algorithmType”);
query.bindEntity(“algorithmType”, algorithmType);
List algorithms = query.list();
```

The above algorithms list will contain as elements the algorithms for that algorithm type.

To sort the above query by the algorithm's code/id:

```
AlgorithmType algorithmType = … ;
Query query = createQuery("from Algorithm algorithm where " + "algorithm.algorithmType = :algorithmType");
query.bindEntity("algorithmType", algorithmType);
query.addResult("algorithm", "algorithm");
query.addResult("algorithmId", "algorithm.id");
query.orderBy("algorithmId", Query.ASCENDING);
List queryResults = query.list();
```

The above queryResults list will contain as elements instances of the interface QueryResultRow. Each query result row will have two values, keyed by "algorithm" and "algorithmId". The list will be ordered (on the database) ascending by the algorithm's IDs.

Since HQL works with the entity's properties instead of the tables' column names, there may be extra research required when writing queries. The source of the property information is in the hibernate mapping document for each entity class- they are documents that exist in the same package as the entity, have the same root file name as the entity's interface, and end with .hbm.xml. These files will give the list of properties available for each entity that can be referenced when writing HQL.

More information can be found in the JavaDocs associated with the Query interface.

Contents

Examples
Union queries
Performance
Raw SQL

6.3.1 Examples

Even with all of the above, there are a few cases that stand out with possibly needing examples in order to help. Notably, dealing with language entries and lookups may be confusing.

Here is an example of selecting all algorithm types where the description is like some input:

```
String likeDescription = …;
Query query = createQuery("from AlgorithmType_Language algTypeLang join algTypeLang.id.parent algType where algTypeLang.description like :likeDescription and algTypeLang.id.language = :language");
query.bindEntity("language", getActiveContextLanguage());
query.addResult("algType", "algType);
query.bindLikableStringProperty("likeDescription", AlgorithmType.properties.languageDescription, likeDescription);
List algorithmTypes = query.list();
```

The algorithmTypes list will contain as elements the algorithm types whose description is like likeDescription. Note that the string likeDescription will have a trailing '%' appended when it is bound to the query.

Here is an example of selecting particular lookup values, with descriptions like an input value:

```
String description = header.getString(STRUCTURE.HEADER.DESCR);
Query query = createQuery("from LookupValue_Language lookupValLang "
        + "where upper(lookupValLang.description) like upper(:description) and lookupValLang.id.language = :language and "
        + "lookupValLang.id.parent.id.fieldName = 'RPT_OPT_FLG');
query.bindLikableStringProperty("description", LookupValue.properties.languageDescription, description);
query.bindEntity("language", getActiveContextLanguage());
query.addResult("lookupValue", "lookupValLang.id.parent");
query.addResult("description", "lookupValLang.description");
query.orderBy("description");
List results = query.list();
```

The list results will contain QueryResultRows, with values keyed by "lookupValue" and "description".

6.3.2 Union queries

You may note that hibernate's HQL does not allow unions, as this does not reconcile with the object oriented approach of HQL. However, as this can be a common technique to apply, a programmatic union has been provided in the Oracle Utilities Framework. The application will actually open two cursors and flip back and forth between rows from each cursor when each would be the next one, based upon the order by clause. This should at most read one extra row from each cursor opened than may be needed (in the case of limited maximum rows).
In order to union two queries, they must have identical result columns, order by clauses, and max rows setting. Note that some of the properties of the union query be modified directly, leaving the individual queries to omit those properties.
Creating a union query is simple. Given two queries that need to be unioned together, simply issue:

```
UnionQuery union = query.unionWith(query2);
```

If a third (or later) query needs to be unioned, add it to the union directly:

```
union.addQueryToUnion(query3);
```

6.3.3 Performance

In order to evaluate the performance of HQL queries, it is necessary to first run the HQL through the hibernate engine at run-time in order to produce the equivalent SQL. First, code the initial HQL into the application or a unit test or standalone executable program. Start the application or test program with SQL tracing turned on. When the HQL under construction executes, grab the SQL from the log/console. Then follow the directions in ??? 07 SQL Programming Standards to check the performance of the SQL.
In general, most of the advice under the SQL programming standards applies equally for coding HQL when applicable at all.

6.3.4 Raw SQL

In rare cases, it may be necessary to forgo the use of HQL and instead use raw SQL. This is not a preferred approach, as the data returned will not be Java entities, but columns of primitive data types. However, for possible performance reasons (no db hints are allowed in HQL) or if a table is not mapped into a Java entity, this approach exists.
There are parallel methods available on subclasses of GenericBusinessObject that create spl PreparedStatements, instead of Query objects. So, instead of createQuery, the method createPreparedStatement should be called on a Raw SQL statement.
The PreparedStatement is similar to the regular jdbc PreparedStatement, but has some extra functionality, and a slightly different interface so that it is similar to the regular HQL Query interface (they are interchangeable in some cases).
The main difference is that the prepared statement is created with raw SQL—use the actual table and column names instead of the Java entity names and property names. Also, the select clause must exist as in normal SQL but not HQL.
Additionally, this break-out into raw SQL allows SQL statements that update table data. Again, this is normally frowned upon, and instead should be done by entity manipulation. However, in cases where a set-based SQL could update many rows at once, this option is available, whereas HQL is ONLY meant for querying without any updates.
For more help on constructing raw SQL queries, please see ??? 07 SQL Programming Standards.

6.4 SQL Programming Standards

Źródło: SQLProgrammingStandards.doc
This document describes the SQL programming standards to be used in any database query. These standards will ensure that all database queries across the system have been structured properly and thus have less chance to cause performance issues. All developers must adhere to these standards.
Contents

Composing SQL Statements
Testing SQL Statements

6.4.1 Composing SQL Statements

Contents

Prerequisite
Composing A SELECT Statement
Existence Checks

6.4.1.1 Prerequisite

This document assumes that you have a basic knowledge of SQL syntax and database functions.

6.4.1.2 Composing A SELECT Statement

Contents

General SELECT Statement Considerations
Selection List
Database-specific Features
FROM Clause
WHERE Clause
Sort Order
Grouping

6.4.1.2.1 General SELECT Statement Considerations

- Before composing an SQL statement, you should have in front of you the ERD of the tables involved in that SQL. You should make sure you fully understand the relationships between the tables.
- As you may know, an SQL may return a single record or a set of records as its result set. When a set is to be returned, it is managed by a cursor that loops through that set and issues a separate database call for each record in the set.

  Therefore, when you design your SQL, think carefully if the task can be easily achieved in a single SQL or rather that the nature of task is such that a row-by-row processing would make more sense. Examples for the latter could be a list processing or simply because the calculation per row is too complicated to be handled by the database.

6.4.1.2.2 Selection List

- If a list of fields is to be returned, specify them prefixed by their table's alias name as specified in the From Clause.
- Use the DISTINCT option when the result list of records may contain duplicate rows in respect to the specified list of fields AND only one copy of the duplicated rows is needed.
- For top-level batch programs, always specify the WITH HOLD keyword on the main SQL of a cursor based processing. This is to keep the cursor open after a commit or rollback. Without this, main cursor will be closed and fetch of the next record or restart processing will fail (specific to DB2) with SQL error 501.

6.4.1.2.3 Database-specific Features

6.4.1.2.3.1 Oracle

- Oracle7 or later provides new approach for optimization: cost-based optimization (CBO). CBO evaluates the cost to, or impact on, your system of the execution path for each specific query and then select the lowest-cost path. The CBO was designed to save you the trouble of fiddling with your queries. Occasionally, it is not giving you the results you want and you have exhausted all other possible problem areas, you can specify hints to direct the CBO as it evaluates a query and creates an execution plan. If you have used hints before, you know that a hint starts with /*+ and ends with */. A hint applies only to the statement in which it resides; nested statements consider as separate statement and require their own hints. Furthermore, a hint currently has a 255-character limit. Since the use of hint is database-specific, we should make use of Database Functions to accomplish it.

- The most effective hints for use with the CBO are:
  - *FULL* – tells the optimizer to perform a full table scan on the table specified in the hint

    SELECT /*+FULL(table_name)*/ COLUMN1,COLUMN2…..
  - *INDEX* – tells the optimizer to use one or more indexes for executing a given query. Note: If you just want to ensure the optimizer doesn't perform a table scan, use INDEX hint without specifying an index name and the optimizer will use the most restrictive index. A specific index should not be used as the actual index name may differ on the client's site.

    SELECT /*+INDEX(table_name index_name1 indexname2…) */ COLUMN1, COLUM2
  - *ORDERED* – tells the optimizer to access tables in particular order, based on the order in the query's FROM clause (often referred to as the driving order for a query)

    SELECT /*+ORDERED*/ COLUMN1, COLUMN2 FROM TABLE1, TABLE2
  - *ALL_ROWS* – tells the optimizer to choose the fastest path for retrieving all the rows of a query, at the cost of retrieving a single row more slowly.

    SELECT /*+ALL_ROWS*/ COLUMN1, COLUMN2…
  - *FIRST_ROWS* – tells the optimizer to choose the approach that returns the first row as quickly as possible. Note: the optimizer will ignore the first rows hint in DELETE and UPDATE statements and in SELECT statements that contain any of the following: set operators, group by clauses, for update clause, group functions, and the distinct operators.

    SELECT /*+FIRST_ROWS*/ COLUMN1, COLUMN2…
  - *USE_NL* – tells the optimizer to use nested loops by using the tables listed in the hint as the inner (non-driving) table of the nested loop. Note: if you use an alias for a table in the statement, the alias name, not the table name, must appear in the hint, or the hint will be ignored.

    SELECT /*+USE_NL(tableA table B) */ COLUMN1, COLUMN2…
- Hints are an Oracle specific feature and are not supported by the DB2 SQL syntax. If you need to add a hint to your SQL make sure that a different SQL version is used for DB2 where the hint is not used.
  Base product developers should not duplicate their SQL in this case but rather use the special database functions file "dbregex.txt". In this file you should add a new hint-code that in Oracle translates into the specific hint whereas in DB2 it translates into an empty string.

6.4.1.2.4 FROM Clause

- Any table that has least one of its fields specified in the Selection List and/or any table that is directly referred to in the Where Clause (excluding sub-selects if any) must be specified in this section.
- Label each table with a meaningful short alias and use this alias to reference the table anywhere in the SQL.

6.4.1.2.5 WHERE Clause

Contents

General WHERE Clause Considerations
Use of Sub-Selects
Use of IN Function
Use Of Database Functions
Other

6.4.1.2.5.1 General WHERE Clause Considerations

- All tables specified in the From Clause must participate in a join statement with another table. Table left not joined, would cause a Cartesian join to be applied for this table and any other table on the list, resulting in an incorrect result list let alone very poor performance.
- Note that there is no such thing as "conditional" join where the only join statement for a table is under a condition. In cases where the condition is not met and thus the join is not performed, one would end up with the same problem described previously.
- The final result set is built up by taking the full population of the tables involved and applying the restricting criteria to it one after another where the intermediate result population of one step is the input for the next step. Therefore, it is recommended to specify the most restrictive criteria first so that at the end of one step, lesser records are processed in the next step.

  This is of course a very schematic and simplified way to describe the internal process. This is not necessarily how the database is actually processing the statements. However, setting up the criteria as described would direct the database to use the right path.

6.4.1.2.5.2 Use of Sub-Selects

- When you need to further test each processed record in the Where clause for meeting an additional condition, AND that condition can NOT be checked directly on the Where clause level, you probably need a sub-select.
- As it is performed once for each outer level record it is considered as quite an expensive tool. Therefore if the criteria checked in a sub-select can be moved to the outer where clause level, it is preferable. If you still need to use a sub-select, it is very important to restrict the outer where clause population to the very minimum possible so that lesser records would need to be further checked for the sub-select condition.
- When no value needs to be returned from the sub-select query but rather simply use it to check if a certain condition is true or false, use the EXISTS function as follows:

  Select …
  From …
  Where … AND EXISTS (<sub-select>)

- A sub-select query may refer to any value of the outer level record as its input parameters. Notice that if your sub-select does NOT refer to any of the processed record fields, it means that the result set of the sub-select would be the same for ALL the processed records.

  Note that this could, but not necessarily, be an indication that your sub-select is set up wrong. One case where it is definitely wrong is when the sub-select result is input to an EXISTS function.

6.4.1.2.5.3 Use of IN Function

- Whenever a field needs to be tested against a list of valid values it is recommended to use the IN function and not compare the field against each and every value.

  Wrong way:

  Select …
  From …
  Where … (A = '10' or A='20' or A='30')

  Right way:

  Select …
  From …
  Where … A IN ('10','20','30')

6.4.1.2.5.4 Use Of Database Functions

- Not all database functions available for one database are valid for others. Make sure that when you do use a database function the SQL works properly on every database supported by the product.
- Avoid using LIKE as this will cause table scans. To achieve the 'LIKE' function where the first part of the string is specified, e.g., "CM%", BETWEEN may be used with the input criteria padded with high and low values.

6.4.1.2.5.5 Other

- Depending on the data distribution, search on optional index column will likely to cause time out. See example –

  Select BSEG_ID
  From CI_BSEG
  Where MASTER_BSEG_ID = &IN.MASTER-BSEG-ID
  For such cases, consider additional restrictions or re-create the index to become composite – MASTER_BSEG_ID + BSEG_ID.

6.4.1.2.6 Sort Order

- When a result list should be displayed in a specific order, sorting should take place on the database level and NOT on the client. This is especially important in cases when the list cannot be returned in full but rather in batches of records. Sorting each batch of records separately would not guarantee the sort order between records of different batches.
- Columns in the sort order list must be specified in the selection list.
- Prefix each field used in this clause with its table's alias name.
- Explicitly specify whether sorting should be ascending or descending and do not rely on database defaults.

6.4.1.2.7 Grouping

- When a set of records needs to be grouped together by a simple and straightforward condition, it is recommended to use the database Group By Clause. In this case only the final summarized records are to be returned to the client resulting in a lesser number of database calls as opposed to processing the full list let alone a simpler program without any special grouping logic.

6.4.1.3 Existence Checks

- The common technique used to check whether a certain condition is met or not, obviously when no data needs to be returned, is simply COUNT how many records match that condition. A zero number indicates that no record has met that condition.

  Notice that this is not very efficient as we are asking the database to scan the records for an accurate number that we don't really care about. All we really want to know if there is at least one such record and NOT how many they are.

  When the tables involved are of low volume there should be no problem using this technique. It is very simple and uses common SQL syntax to all databases.

- However, when that condition is checked against a high volume table that many of its records meet that condition, scanning all the matching records to get a count we don't need should be avoided.

  In this case use the EXISTS function as follows:

  ```
  Select 'x'
  From <The main table of the searched field, where it is defined as the PK of that table>
  Where <search field> = <search value> and
        EXISTS
     (<sub-select with the desired condition. This is the high volume table>);
  ```

  For example :

  ```
  Select 'x'
  From CI_UOM
  Where UOM_CD = input UOM_CD and
        EXISTS (select 'x'
               From CI_BSEG_CALC_LN
               Where UOM_CD = input UOM_CD);
  ```

  If this does not work for your special case, use the following option :

  ```
  Select 'x'
  From CI_INSTALLATION
  Where EXISTS
      (<sub-select with the desired condition>) ;
  ```

  Remember : This type of existence check using the Installation Options record should only be used in rare cases and should be consulted with the DBA first before implementation.

  Note that we use CI_INSTALLATION as this table has only one row.

6.4.2 Testing SQL Statements

Contents

Result Data
Performance Testing – Oracle Only
More Extensive Performance Testing

6.4.2.1 Result Data

Once your SQL is ready, it is essential to test that it actually returns the expected result.
Create sample data for each condition checked by your SQL. Then execute the SQL and make sure it returns the expected result for each case.

6.4.2.2 Performance Testing – Oracle Only

Contents

Overview
What Is An Explain Plan?
Generate The SQL's Explain Plan
Analyze Explain Plan

6.4.2.3 Overview

An SQL may perform reasonably well even if not efficiently written in cases where the volume of processed data is low, like in a development environment. However, the same SQL may perform very poorly when executed in a real high volume environment. Therefore, any SQL should be carefully checked to make sure it would provide reasonable performance at execution time.
Obviously there could be many reasons for an SQL to perform poorly and not all of them are easy to predict or track.
In general, these could be subcategorized into two main groups:

- Basic issues related to the SQL code. These may be missing JOIN statements, inefficient path to the desired data, inefficient use of database functions, etc.
- More complicated issues having to do with lack of indexes, database tuning and handling of high volume of data, efficiency of I/O system etc.

The latter group of issues may only be truly tested on a designated environment simulating a real production configuration. These performance tests are typically conducted by a team of database and operating system experts as part of a thorough performance testing of a predefined set of process.
It is the first group of issues that can and should be tested by the programmer at this stage. This is done by analysis of the SQL's ***Explain Plan*** result.

6.4.2.4 What Is An Explain Plan?

An explain plan is a representation of the access path that is taken when an SQL is executed within Oracle.
The optimal access path for an SQL is determined by the database optimizer component. With the ***Rule Based Optimizer*** (RBO) it uses a set of heuristics to determine access path. With the ***Cost Based Optimizer*** (CBO) we use statistics to analyze the relative costs of accessing objects.
Since the Cost Based optimizer relies on actual data volume statistics to determine the access path, to generate an accurate Explain Plan using the cost based optimizer requires a database set up with the proper statistics of a real high volume data environment.

Note. A cost based optimizer Explain Plan generated on an inadequate database, would be totally inaccurate and misleading!

Obviously, our development database does not qualify as an optimal environment of cost based optimizations. Since the Rule Based optimizer is not data dependant it would provide a more reliable Explain Plan for this database.

Note. An efficient rule based Explain Plan does not guarantee an efficient cost based one when the SQL is finally executed on the real target database. However, a poor rule based Explain Plan would most probably remain such on a database with a higher volume of data.

Note. When the SQL is complicated and mainly designed to process high volume tables it is recommended to also analyze its Explain Plan on an appropriate high volume database.

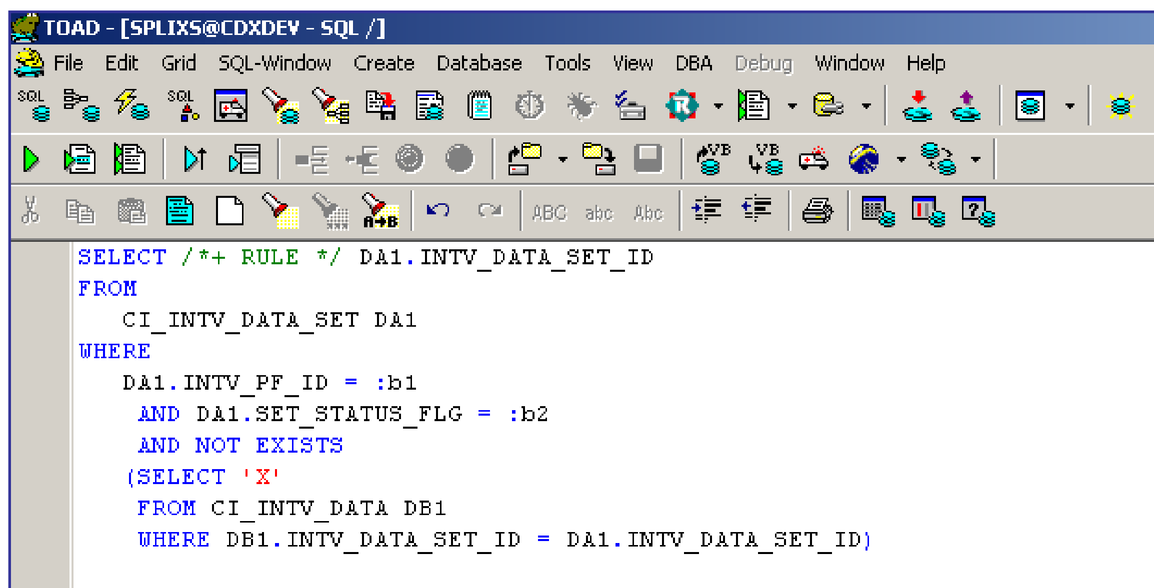
6.4.2.4.1 Generate The SQL's Explain Plan

- Let's assume this is the SQL to be checked

```
SELECT
   DA1.INTV_DATA_SET_ID
FROM
   CI_INTV_DATA_SET DA1
WHERE
   DA1.INTV_PF_ID = :S-ERRDS-IN-DATA.RES-INTV-PF-ID
    AND DA1.SET_STATUS_FLG =
:S-ERRDS-IN-DATA.ERROR-STATUS-FLG
    AND NOT EXISTS
   (SELECT 'X'
    FROM CI_INTV_DATA DB1
    WHERE DB1.INTV_DATA_SET_ID = DA1.INTV_DATA_SET_ID)
```

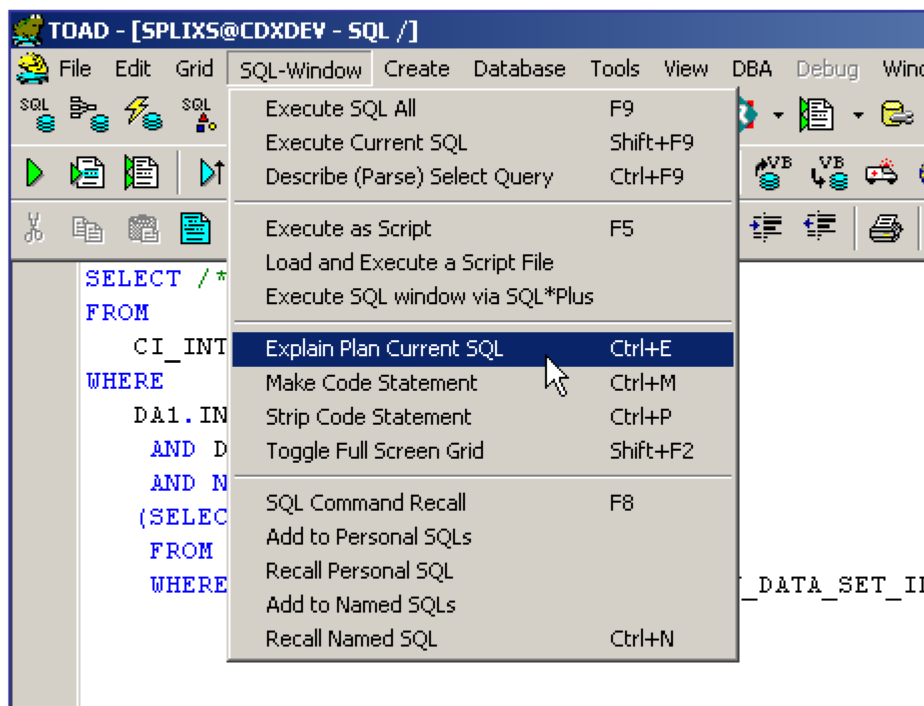
SQL To Check

- Adjust SQL Statement:
    - Extract the tested SQL into Toad SQL editor.
    - Replace the COBOL name of each ***Host Variable*** with the equivalent database identifier ***:b<n>*** where n is a unique number identifying that host variable. If the same variable appears more than one in the SQL use the same database host variable id in all occurrences.
    - Force the database to analyze the SQL in ***Rule Base*** mode by introducing the RULE database hint phrase.



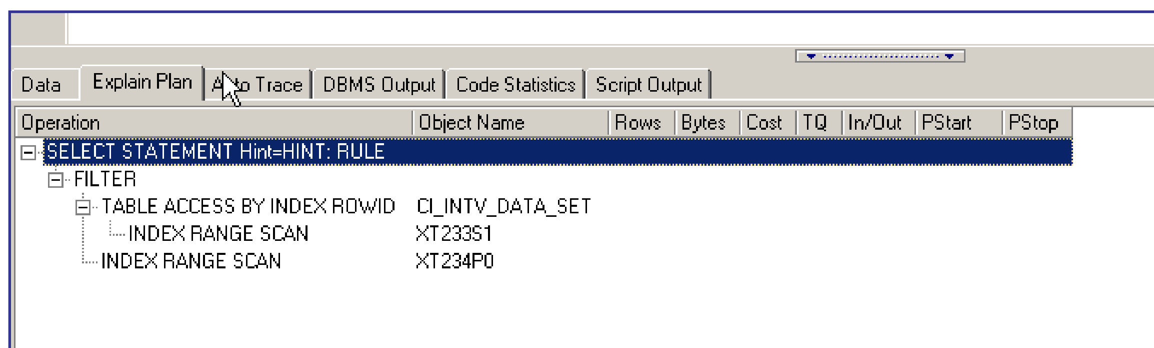
Adjust SQL Statement

- Generate the ***Explain Plan:***
    - Position the cursor on the SQL area
    - Choose the "Explain Plan Current SQL" option on the SQL-Window menu. Alternatively, use CTRL+E.



Get Explain Plan

- The generated plan appears on the result section of the editor.



Explain Plan

6.4.2.4.2 Analyze Explain Plan

Contents

Access Methods

Common Issues To Be Aware Of

6.4.2.4.2.1 Access Methods

Logically Oracle finds the data to read by using the following methods:

- ***Full Table Scan*** (FTS). Using this method the whole table is read.
- ***Index Lookup*** (unique & non-unique). Using this method, data is accessed by looking up key values in an index and returning rowids where a rowid uniquely identifies an individual row in a particular data block.
- ***Rowid.*** This is the quickest access method available Oracle simply retrieves the block specified and extracts the rows it is interested in. Most frequently seen in explain plans as Table access by Rowid.

6.4.2.4.2.2 Common Issues To Be Aware Of

Contents

Cartesian Product
Full Table Scan
Join Order
Nested Loops
Sort

6.4.2.4.2.2.1 Cartesian Product

- A Join is a predicate that attempts to combine 2 row sources. Cartesian Product is created when there are no join conditions between 2 row sources and there is no alternative method of accessing the data. Typically this is caused by a coding mistake where a join has been left out. The CARTESIAN keyword in the Explain Plan indicates this situation.

6.4.2.4.2.2.2 Full Table Scan

- A Full Table Scan, i.e. TABLE ACCESS FULL phrase, found in the Explain Plan usually indicates an inefficient access path. This means that the only way the database found to get to the desired data is by reading every single row in the table.

  Notice that if the logic indeed requires reading all data, then this database decision is indeed correct. However, if you intended to get a small subset of rows from a large table and ended up reading all of it this is definitely not efficient and should be fixed. If this is the case, try and find a better SQL structure that would avoid a full table access. If you can't find such, please consult a DBA as this SQL may require an additional Index to be created for the table.

- Sometimes there would be a proper index on a particular table but still a full table scan would be chosen for the access path of that table. This may be as result of an inefficient Join Order. Please see details below.

6.4.2.4.2.2.3 Join Order

A Join is a predicate that attempts to combine 2 row sources. We only ever join 2 row sources together. Join steps are always performed serially even though underlying row sources may have been accessed in parallel. The join order makes a significant difference to the way in which the query is executed. By accessing particular row sources first, certain predicates may be satisfied that are not satisfied by with other join orders. This may prevent certain access paths from being taken.

- Make sure the join between 2 tables is done via indexed fields as much as possible.
- Also, if such an index exists, make sure you specify fields in the order they are defined by that index.

6.4.2.4.2.2.4 Nested Loops

This is a common type of processing a join between 2 row sources. First we return all the rows from row source 1, then we probe row source 2 once for each row returned from row source 1.

```
Row source 1
Row 1 --------------    -- Probe ->    Row source 2
Row 2 --------------    -- Probe ->    Row source 2
Row 3 --------------    -- Probe ->    Row source 2
```

Row source 1 is known as the ***outer table.*** Row source 2 is known as the ***inner table.*** Accessing row source 2 is known a probing the inner table. For nested loops to be efficient it is important that the first row source returns as few rows as possible as this directly controls the number of probes of the second row source. Also it helps if the access method for row source 2 is efficient as this operation is being repeated once for every row returned by row source 1.

6.4.2.4.2.2.5 Sort

Sorts are expensive operations especially on large tables where the rows do not fit in memory and spill to disk. There are a number of different operations that promote sorts:

- Order by clauses
- Group by

- Sort merge join

Note that if the row source is already appropriately sorted then no sorting is required. In other words, if the fields you sort by happen to be defined by an Index in that particular order then sort operation is avoided. Therefore, whenever you see that an explicit sort operation has taken place, check if it can be avoided by using an index or sometimes just by making sure your are using an index's fields in the right order.
If no such index exists and the number of rows to be sorted is of high volume, please consult a DBA as this may justify adding a new index.

6.4.2.5 More Extensive Performance Testing

Special attention should be paid to background processes that are designed to process high volume tables. A thorough performance testing exercise in a benchmark format may be called upon.

6.5 Database Design Standards

Źródło: DatabaseDesignStandards

The objective of this document is to provide a standard for database objects (such as tables, columns, and indexes) for products using Oracle Utilities Framework. This standard is introduced to insure clean database design, to promote communications, and to reduce errors leading to smooth integration and upgrade processes. Just as Oracle Utilities Framework goes thorough innovation in every release of the software, it is also inevitable that the product will take advantage of various database vendors' new features in each release. The recommendations in the database installation section include only the ones that have been proved by vigorous QA processes, field tests and benchmarks.

6.5.1 Database Object Standard

This section discusses the rules applied to naming database objects and the attributes that are associated with these objects.

Contents

Naming Standards
Column Data Type and Constraints
Standard Columns

6.5.2 Naming Standards

The following naming standards must be applied to database objects.

Contents

Table
Columns
Indexes
Sequence
Trigger

6.5.2.1 Table

Table names are prefixed with the owner flag value of the product. For customer modification **CM** must prefix the table name. The length of the table names must be less than or equal to 30 characters. A language table should be named by suffixing **_L** to the main table. The key table name should be named by suffixing **_K** to the main table. It is recommended to start a table name with the 2-3 letter acronym of the subsystem name that the table belongs to. For example, **MD** stands for meta-data subsystem and all meta data table names start with **CI_MD**.

Some examples are:

- CI_ADJ_TYPE
- CI_ADJ_TYPE_L

> A language table stores language sensitive columns such as a description of a code. The primary key of a language table consists of the primary key of the code table plus language code (LANGAGUE_CD).

A key table accompanies a table with a surrogate key column. A key value is stored with the environment id that the key value resides in the key table.

The tables prior to V2.0.0 are prefixed with CI_ or SC_.

6.5.2.2 Columns

The length of a column name must be less than or equal to 30 characters. The following conventions apply when you define special types of columns in the database.

- Use the suffix **FLG** to define a lookup table field. Flag columns must be CHAR(4). Choose lookup field names carefully as these column names are defined in the lookup table (CI_LOOKUP_FLD) and must be prefixed by the product owner flag value.
- Use the suffix **CD** to define user-defined codes. User-defined codes are primarily found as the key column of the admin tables.
- Use the suffix **ID** to define system assigned key columns.
- Use the suffix **SW** to define Boolean columns. The valid values of the switches are 'Y' or 'N'. The switch columns must be CHAR(1)
- Use the suffix **DT** to define Date columns.
- Use the suffix **DTTM** to define Date Time columns.
- Use the suffix **TM** to define Time columns.

Some examples are:

- ADJ_STATUS_FLG
- CAN_RSN_CD

6.5.2.3 Indexes

Index names are composed of the following parts:

[X][C/M/T]NNN[P/S]

- **X** – letter **X** is used as a leading character of all base index names prior to Version 2.0.0. Now the first character of product owner flag value should be used instead of letter X. For client specific implementation index in **Oracle**, use **CM.**
- **C/M/T** – The second character can be either **C** or **M** or **T**. **C** is used for control tables (Admin tables). **M** is for the master tables. **T** is reserved for the transaction tables.
- **NNN** – A three-digit number that uniquely identifies the table on which the index is defined.
- **P/S/C** – **P** indicates that this index is the primary key index. **S** is used for indexes other than primary keys. Use **C** to indicate a client specific implementation index in **DB2** implementation.

Some examples are:

- XC001P0
- XT206S1
- XT206C2
- CM206S2

Warning! Do not use index names in the application as the names can change due to unforeseeable reasons.

6.5.2.4 Sequence

The base sequence name must be prefixed with the owner flag value of the product.

6.5.2.5 Trigger

The base trigger name must be prefixed with the owner flag value of the product.

When implementers add database objects, such as tables, triggers and sequences, the name of the objects should be prefixed by **CM.** For example, Index names in base product are prefixed by **X;** the Implementers' index name must not be prefixed with **X.**

6.5.2.6 Column Data Type and Constraints

Contents

User Define Code
System Assigned Identifier
Date/Time/Timestamp
Number
Fixed Length/Variable Length Character Columns
Null Constraints
Default Value Setting
Foreign Key Constraints

6.5.2.6.1 User Define Code

User Defined Codes are defined as CHAR type. The length can vary by the business requirements but a minimum of eight characters is recommended. You will find columns defined in less than eight characters but with internationalization in mind new columns should be defined as CHAR(10) or CHAR(12). Also note that when the code is referenced in the application the descriptions are shown to users in most cases.

6.5.2.6.2 System Assigned Identifier

System assigned random numbers is defined as CHAR type. The length of the column varies to meet the business requirements. Number type key columns are used when a sequential key assignment is allowed or number type is required to interface with external software. For example, Notification Upload Staging ID is a Number type because most EDI software uses a sequential key assignment mechanism. For sequential key assignment implementation, the DBMS sequence generator is used in conjunction with Number Type ID columns.

6.5.2.6.3 Date/Time/Timestamp

Date, Time and Timestamp columns are defined physically as DATE in Oracle. In DB2 the DATE, TIME and TIMESTAMP column types, respectively, are used to implement them. Non-null constraints are implemented only for the required columns.

6.5.2.6.4 Number

Numeric columns are implemented as NUMBER type in Oracle and DECIMAL type in DB2. The precision of the number should always be defined. The scale of the number might be defined. Non-null constraints are implemented for all number columns.

6.5.2.6.5 Fixed Length/Variable Length Character Columns

When a character column is a part of the primary key of a table define the column in CHAR type. For the non-key character columns, the length should be the defining factor. If the column length should be greater than 10, use VARCHAR2 type in Oracle and VARCHAR type in DB2.

6.5.2.6.6 Null Constraints

The Non-null constraints are implemented for all columns except optional DATE, TIME or TIMESTAMP columns.

6.5.2.6.7 Default Value Setting

The rule to set the database default value is the following:

- When a predefined default value is not available, set the default value of Non-null CHAR or VARCHAR columns to blank except the primary key columns.

- When a predefined default value is not available, set the default value Non-null Number columns to 0 (zero) except the primary key columns.
- No database default values should be assigned to the Non Null Date, Time, and Timestamp columns.

6.5.2.6.8 Foreign Key Constraints

Referential Integrity is enforced by the application. In database, do not define FK constraints. Indexes are created on most of Foreign Key columns to increase performance.

6.5.2.7 Standard Columns

Contents
  Owner Flag
  Version

6.5.2.7.1 Owner Flag

Owner Flag (OWNER_FLG) columns exist on the system tables that are shared by multiple products. Oracle Utilities Framework limits the data modification of the tables that have owner flag to the data owned by the product.

6.5.2.7.2 Version

The Version column is used to for optimistic concurrency control in the application code. Add the Version column to all tables that are maintained by a Row Maintenance program irrespective of the language used (COBOL or JAVA).

6.6 Numeracja więzów integralności

Niniejsza strona zawiera informacje o numerowaniu więzów integralności w zależności od tego jak powstają (import/tworzenie).

6.6.1 Nowe więzy integralności

Tworząc nowe więzy integralności kierujemy się następującą konwencją:

CM2<xxx>[P|F]<n>

,gdzie:
<xxx> - 3-znakowy numer constraint'a,
[P|F] - P-klucz główny, F-klucz obcy
<n> - kolejny numer constraint'a

Tak więc wartość numeryczną rozpoczynamy od wartości **2**.

Przykład:
CM2100P0 - dla klucza głównego
CM2100F0 - dla klucza obcego
CM2355F0 - dla klucza obcego (1. klucz obcy dla tabeli)
CM2355F1 - dla klucza obcego (2. klucz obcy dla tabeli)

Chcąc szybko wyznaczyć numer kolejnego constraint'a możemy posłużyć się poniższym zapytaniem (**tylko dla zrealizowanych tabel**):

```
SELECT   table_name, constraint_name, constraint_type
  FROM   all_constraints
 WHERE   owner = 'CISADM' AND CONSTRAINT_NAME LIKE 'CM2%'
```

ORDER BY   constraint_name

6.6.2 UWAGA

Należy pamiętać, że na poziomie bazy danych zakładamy wyłącznie więzy integralności kluczy głównych! Natomiast oba typy więzów integralności (klucze główne, klucze obce) definiujemy później w formie metadanych w aplikacji CCB.

6.7 Zmiany w schemacie bazy danych

Modyfikacje w systemie CC&B wymagać mogą zmian w schemacie bazy danych jak też w danych konfiguracyjnych systemu. Zmiany te muszą wyjść w paczce z oprogramowaniem - stąd muszą być śledzone i utrzymywane na repozytorium.

6.7.1 Lokalizacja

Zmiany takie jak dodanie nowej tabeli, nowy constraint, indeks wrzucamy do katalogu: $SPLEBASE/db/changes

6.7.2 Format nazwy pliku

Zmiany wprowadzamy jako skrypt SQL. Format nazwy pliku:

<DATA>_SR-<NR_REDMINE>_<NR_PORZĄDKOWY>_<DODATKOWE_INFO>.sql

gdzie:
<DATA> - data wprowadzenia zmiany w formacie YYYYMMDD
<NR_REDMINE> - numer zgłoszenia redmine dotyczące tej zmiany
<NR_PORZĄDKOWY> - (opcjonalny) wymagany do określenia porządku kiedy mamy kilka zmian do jednego zgłoszenia. Określa on porządek wykonania skryptów na środowisku instalacyjnym.
<DODATKOWE_INFO> - (opcjonalny) dodatkowe informacje

Przykład:
20130219_SR-734_1.sql

6.8 Zmiany w danych konfiguracyjnych aplikacji

Obiekty konfiguracyjne, które tworzymy w systemie CC&B aby trafiły wraz z kodem na kolejne środowiska należy umieścić w repozytorium. Obiekty te muszą być dostarczane w formie kompletnych obiektów MO (Maintenance Object) poprzez pliki XML. Przygotowane pliki XML muszą być umieszczone w repozytorium w odpowiednim katalogu:

- $SPLEBASE\db\ccb_mo_rep – konfiguracja techniczna
- $SPLEBASE\db\ccb_mo_rep_bc_eob – konfiguracja biznesowa EOB
- $SPLEBASE\db\ccb_mo_rep_bc_eop – konfiguracja biznesowa EOP

Wewnątrz powyższych katalogów znajdują się pliki XML pogrupowane w katalogi per obiekt MO (nazwa MO jest nazwą katalogu).

6.8.1 Przykładowy plik XML

```xml
<?xml version="1.0" encoding="UTF-8"?>
<moData mo="ALG TYPE" dontDelete="false" onlyInsert="false" forceUpsert="false">
  <table name="CI_ALG_TYPE" where="WHERE  ALG_TYPE_CD IN('CMCHVALID' )">
    <row>
      <ALG_TYPE_CD sqlType="CHAR" isPK="Y">CMCHVALID   </ALG_TYPE_CD>
      <PGM_NAME sqlType="VARCHAR2"> </PGM_NAME>
      <ALG_ENTITY_FLG sqlType="CHAR">ZCHV</ALG_ENTITY_FLG>
      <VERSION sqlType="NUMBER">2</VERSION>
      <OWNER_FLG sqlType="CHAR">CM  </OWNER_FLG>
      <PGM_TYPE_FLG sqlType="CHAR">PLSR</PGM_TYPE_FLG>
      <SCR_CD sqlType="CHAR">CMCHVAL     </SCR_CD>
      <table name="CI_ALG_TYPE_L" where="WHERE  ALG_TYPE_CD IN('CMCHVALID   ' )">
        <row>
          <ALG_TYPE_CD sqlType="CHAR" isPK="Y">CMCHVALID   </ALG_TYPE_CD>
          <LANGUAGE_CD sqlType="CHAR" isPK="Y">ENG</LANGUAGE_CD>
          <DESCR50 sqlType="VARCHAR2">Sprawdzenie wartości ad-hoc na charakterystyce</DESCR50>
          <DESCRLONG sqlType="VARCHAR2">Sprawdzenie wartości ad-hoc na charakterystyce. Jeżeli wartość będzie niepoprawna zostanie zgłoszony błąd.</DESCRLONG>
          <VERSION sqlType="NUMBER">1</VERSION>
          <OWNER_FLG sqlType="CHAR">CM  </OWNER_FLG>
        </row>
      </table>
      <table name="CI_ALG_TYPE_PRM" where="WHERE  ALG_TYPE_CD IN('CMCHVALID   ' )">
        <row>
          <ALG_TYPE_CD sqlType="CHAR" isPK="Y">CMCHVALID   </ALG_TYPE_CD>
          <SEQNO sqlType="NUMBER" isPK="Y">10</SEQNO>
          <VERSION sqlType="NUMBER">2</VERSION>
          <PARM_REQ_SW sqlType="CHAR">Y</PARM_REQ_SW>
          <OWNER_FLG sqlType="CHAR">CM  </OWNER_FLG>
          <table name="CI_ALG_TYPE_PRM_L" where="WHERE  ALG_TYPE_CD IN('CMCHVALID   ' ) AND SEQNO IN('10' )">
            <row>
              <ALG_TYPE_CD sqlType="CHAR" isPK="Y">CMCHVALID   </ALG_TYPE_CD>
              <SEQNO sqlType="NUMBER" isPK="Y">10</SEQNO>
              <LANGUAGE_CD sqlType="CHAR" isPK="Y">ENG</LANGUAGE_CD>
              <ALG_PARM_LBL sqlType="VARCHAR2">Wartość wyrażenia regularnego do walidacji wartośći charakterystyki</ALG_PARM_LBL>
              <VERSION sqlType="NUMBER">1</VERSION>
              <OWNER_FLG sqlType="CHAR">CM  </OWNER_FLG>
            </row>
          </table>
        </row>
      </table>
    </row>
  </table>
</moData>
```

6.9 Nazewnictwo obiektów w bazie danych

6.9.1 Tabela

CM_ SKRÓT_OKREŚLAJĄCY_MODUŁ *nazwa*

np:

- CM_MSG (tabela główna do komunikatów)
- CM_MSG_RECP (tabela z odbiorcami komunikatów)

W przypadku, gdy potrzebne są tabele z kluczami i tłumaczeniami stosujemy standardowe zakończenie nazw, odpowiednio: **_K** i **_L**

Uwaga: Tabel z _K jeżeli nie ma uzasadnionej potrzeby to nie tworzymy. W 2.4 odchodzą od mechanizmu archiwizacji danych opierającego się na rozdzielaniu środowisk i tabelach kluczy _K

6.9.2 Kolumny w tabeli

Jeżeli korzystamy z bazowych (istniejących) w systemie kolumn - nazwa jak i typ bierzemy tak jak jest w metadanych CCB.

Jeżeli tworzymy nowe pole to:

- nazwa zaczyna się standardowo od **CM_**.
- nazwa pola ma max. 30 znaków
- wszystkie znaki są UPPERCASE
- wyrazy oddzielamy znakiem "_" (tz. podkreślenie/"kładka")

W zależności od przeznaczenia pola używamy odpowiednich suffixów:

- **_FLG** - dla pól które będą zasłowikowane obiektem Lookup. Typ tego pola jest wtedy CHAR(4)
- **_ID** - dla pól klucza które przechowują identyfikatory wygenerowane przez system. Typ pola CHAR(8-max.15)
- **_CD** - dla pól klucza które są utworzone przez użytkownika. Kod - np. Typ Wpisu w Archiwum, np. CM_ARCH_ENTRY_TYPE_CD - CHAR(30)
- **_SW** - dla pól Boolean. Typ: CHAR(1). Wartości **Y** lub **N**.
- **_DT** - dla pól daty
- **_DTTM** - dla pól daty i czasu

6.9.3 Klcze PK/FK

Opisano w sekcji „Numeracja więzów integralności”.

6.9.4 Indeksy

- max. 12 znaków
- każda tabela ma unikalny numer *XXX*, który powtarza się w nazwach jej więzów integralności - CMXXX
- CMXXXS0 ... CMXXXS9

Przykład:

- CM201S0
- CM201S1

6.10 Nazewnictwo obiektów w aplikacji

Uwagi ogólne:

- Znaki **pogrubione** oznaczają stały fragment
- Znaki podkreślone oznaczają zmienną część (np. skrót/nazwę modułu)
- Znaki *pochylone* oznaczają nazwę właściwą.
- Nazwy są **BEZ SPACJI**.

6.10.1 Metadane obiektów DB: Field, Table, Constraint

Obiekty te są odzwierciedleniem obiektów utworzonych w schemacie bazy danych CCB. Stąd konwencje nazewnicze są takie same jak dla obiektów bazy danych. Te są opisane w sekcji „Nazewnictwo obiektów w bazie danych".

6.10.2 Pola - Etykiety (max 30 znaków)

Fikcyjne pole to takie, które nie jest składnikiem żadnej tabeli. Najczęściej wykorzystywane w celu uzyskania etykiety, która może być bez problemu potem użyta na formatkach i innych miejscach CCB, gdyż jest standardowo obsługiwana.
Stosujemy konwencję, że zawsze poprzedzamy prefixem **CM_** i kończymy postfixem **_LBL**. W środku oczywiście może się znajdować określenie modułu, jeżeli jest to label specyficzny tylko dla niego. Przykładowo:

- CM_MSG_RECPIENTS_LBL (odbiorcy komunikatów, z prefixem modułu)
- CM_COUNTER_LBL (ogólna etykieta dla licznika, bez precyzowania czego dotyczy)

6.10.3 Maintenance object (max 12 znaków)

- prefix: **CM-**
- nazywamy UPPERCASE
- max
- mogą występować spacje

Przykłady:

- CM-EMAILTEMP
- CM-ORGTYPE
- CM-RSCHAR

6.10.4 Business object (max 30 znaków)

- prefix: **Cm** (bez "-" lub "_")
- nazywamy notacją wielbłądzią

Przykłady:

- CmArchiveEntryType
- CmPartnerOrganizationUnit

6.10.5 Business Service (max 30 znaków)

patrz: Business Object

6.10.6 Script (max 12 znaków)

- prefix **Cm** (bez "-" lub "_")

- nazywamy notacją wielbłądzią
- skrypty typu Maintenance dla BO nazywamy: **CmXXXXXXMain**, gdzie XXXXXX to nazwa obiektu obsługi

Przykłady:

- CmArchUplFil
- CmArchEntRel
- CmRSCharEdit

6.10.7 Portal (max 8 znaków)

- UPPERCASE
- prefix **CM-**
- Wszystko piszemy ciągiem

6.10.8 UI Map (max 30 znaków)

Patrz: Business Object

6.10.9 Zone (max 12 znaków)

- UPPERCASE
- prefix **CM-**
- Wszystko piszemy ciągiem
- jeżeli jest to Zone która wykonywana jest w celu pobrania jakiejś wartości, prefix: **CMGET**
- jeżeli Zona jest Multi Query to dodajemy - suffix **MQ**
- jeżeli jest to Zone która jest wyszukiwaniem - suffix **Q**, jeżeli jest to wyszukanie jedno z wielu **Q1**,*Q2*,...

Przykłady:

- CM-GETICOURL
- CM-RATESCHQ2
- CM-BILLMQ

6.10.10 Algorithm / Algorithm Type (max 12 znaków)

- UPPERCASE
- prefix **CM-**

6.10.11 Feature Configuration (max 12 znaków)

- UPPERCASE
- prefix **CM-**

6.10.12 Constraint

- max. 12 znaków
- każda tabela ma unikalny numer **XXX**, który powtarza się w nazwach jej więzów integralności - **CMXXX**

6.10.12.1 Primary Key

- CMXXXP0

Przykład:

- CM201P0

6.10.12.2 Foreign Key

- CMXXXF0 ... CMXXXF9

Przykład:

- CM217F0
- CM217F1

6.11 Struktura katalogów w repozytorium

Szeroko rozumiane oprogramowanie wytwarzane przez Dostawcę powinno być umieszczane w prawidłowych lokalizacjach struktury katalogów repozytorium. Poniżej przedstawiona została sugerowana struktura folderów wraz z opisem wskazującym na to, co poszczególne foldery zawierają.

Item	Directory	Content
Java sources	java\source\cm\com\splwg\cm.domain	Java source code.
AppViewer files	splapp\applications\appViewer\data\source\CM	App viewer source code files.
	splapp\applications\appViewer\data\xml\CM	App viewer XML files.
Web Application	splapp\applications\root\cm	UI code.
	splapp\applications\root\WEB-INF\lib	cm.jar file is deployed here.
XAI Application	splapp\applications\XAIApp\WEB-INF\lib	CM?*.jar file. cm.jar file is deployed here.
MPL Application	splapp\mpl\lib	CM?*.jar file.
Standalone Application	splapp\standalone\\lib	CM?*.jar file. cm.jar file is deployed here.
Help files	splapp\applications\help\XXX\cm	Help files segregated by language.
XAI Schemas	splapp\xai\schemas\CM*.xml	XAI schemas.
Service XML files	splapp\xmlMetaInfo\CM*.xml	Service XMLs for Java.

Ponadto:
$SPLEBASE\db\ccb_mo_rep – pliki XML z konfiguracją techniczną
$SPLEBASE\db\ccb_mo_rep_bc_eob – pliki XML z konfiguracją biznesową EOB
$SPLEBASE\db\ccb_mo_rep_bc_eop – pliki XML z konfiguracją biznesową EOP
$SPLEBASE\db\changes – skrypty SQL lub PL/SQL modyfikujące strukturę lub dane w bazie danych
$SPLEBASE\cobol\source\cm – kod źródłowy COBOL dla modyfikacji klienta
$SPLEBASE\runtime – kompilaty COBOL (dll oraz so) dla modyfikacji klienta
$SPLEBASE\structures – struktury menu dla narzędzia konfiuracji środowiska
$SPLEBASE\templates – szablony plików konfiguracyjnych

$SPLEBASE\perl\integration\interfaces – interfejsy rozwiązania integracyjnego perl