# KENTOR PL\*SQL standard och instrumentering

**Summering**

Det här dokumentet beskriver hur PL\*SQL koden skall skrivas och instrumenteras i projekt som använder den här standarden.

Meningen med en standard är att den skall vara tvingande i det den föreskriver. Alternativet är rekommendationer och då inget annat anges så är allt i det här dokumentet en standard.

Dokumentet kan tyckas styra mycket, men att bestämma hur saker skall namnges eller formateras gör det lättare att underhålla och är inte det man bör ägna tid åt i varje enskilt program.

**Paketering av kod**

All kod placeras i paket, det skall inte finnas några funktioner eller procedurer som inte ligger i paket.

Anledningen är att kod i paket naturligt organiseras efter funktionellt område, medan fristående procedurer och funktioner fort blir en svårorganiserad samling kod som bara blir svårare och svårare att underhålla. Underhåll av sådan kod skalar helt enkelt inte i takt med att ett system växer.

Paket skall också organiseras efter funktionalitet, inte efter process eller teknisk typ av lösning. Det betyder att kod skall organiseras efter att ha att göra med inlämnings-uppgift, inte efter inläsning och heller inte efter ifall den har att göra med filhantering. Det är dock vettigt att dela upp paketerad kod per funktionellt område i funktioner som uppdaterar data och kod som inte gör det. Anledningen är att behörighet till ett paket ges på paketnivå och dessutom bara ifall man får köra det eller inte.

Namnsättning som skiljer på det behövs förstås och där skall paket som kan uppdatera ha ett tillägg av \_RW medan kod som endast läser skall ha \_RD.

**Dokumentering i kod**

All dokumentering i kod sker på engelska. Skälet är att ett enhetligt språk skall eftersträvas och planering för framtida underhåll med resurser med bristfällig eller icke existerande kunskap i svenska.

Begrepp bör i möjlig mån vara baserade på engelska uttryck. Det undantag som finns är då ett vedertaget begrepp finns och översättning inte tagits fram än.

Då rimlig översättning saknas så skall uttryck skrivas så som det stavas med å och ä som a och ö som o. Inga omskrivningar som aa, ae, och oe skall användas utom då de redan finns i koden. Även då det använts tidigare bör sådana omskrivningar sluta användas. De gör det inte lättare att läsa koden, eftersom de är mer fonetiska i sin natur.

Kommentarer skrivs inom ett scope och inte före. T ex så skrivs en funktion i ett paket ofta så här:

/\*--------------------------------------------------------

PROCEDURE: timestamp

Skapad: Christer Karlsson, Kentor AB, 1999-10-20

Beskrivning: Loggar meddelande

----------------------------------------------------------\*/

PROCEDURE TIMESTAMP (thecode IN VARCHAR2) IS

BEGIN

null;

END;

Varken kommentar eller kod skriven så den följer standarden, men den är med för att visa före och efter just för kommentaren för proceduren.

Kommentaren skall flyttas så att proceduren ser ut så här:

PROCEDURE TIMESTAMP (thecode IN VARCHAR2) IS

/\*--------------------------------------------------------

PROCEDURE: timestamp

Skapad: Christer Karlsson, Kentor AB, 1999-10-20

Beskrivning: Loggar meddelande

----------------------------------------------------------\*/

BEGIN

null;

END;

Vad gör det för skillnad vilken sida om signaturen kommentaren ligger? Rent tekniskt spelar det ingen som helst roll, men det gör en skillnad i hur smidigt det är att redigera koden.

En modern kodeditor har funktioner för “code collapse” dvs, visa inte koden för ett visst block (såsom en funktion eller procedur) utan visa bara första raden och tillåt utvecklaren att växla mellan att se hela blocket och bara första raden.

Genom att lägga kommentaren efter signaturen så göms även kommentaren då proceduren kollapsas. Utan det så blir kod där man kollapsat många procedurer ofta en lång sida med kommentarer som är i vägen då de inte har att göra med det man faktiskt ville se.

Samma teknik för kommentarer bör användas för andra strukturer som skapar block i kod. Dvs, begin och end, if och end if, loop och end loop och så vidare. Det är en aning ovant att läsa kommentarer så till en början, men fördelen med det visar sig mycket fort.

**Anropa procedur/funktion**

En av de viktigaste delarna i välmodulerad PL-kod är förstås att anropen på andra funktioner skrivs så att man från anropet får ut så mycket information som möjligt så att man inte behöver gå till varje funktion och kolla utan att man kan fortsätta läsa den kod man är intresserad av.

En sak som underlättar väldig mycket är att alltid använda “named parameters”. Ett vanligt anrop skulle kunna se ut så här:

mypkg.myproc(v\_1, v\_2, v\_3);

Här vet man inte något om vilka värden som skickas in, vad processen gör med dem och vad man väntar sig för effekt. Givetvis kan namnet på processen och variablerna ofta ge en god indikation men sällan tillräckligt. Skriver man det istället med “named parameters” så skulle det kunna se ut så här.

mypkg.myproc(p\_in\_customer\_number => v\_1  
 ,p\_in\_invoice\_year => v\_2  
 ,p\_out\_total\_invoiced\_amount => v\_3);

Här syns det tydligare att man skickar in ett kundnummer och ett faktureringsår, tillbaka får man (i v\_3) en summa på hur mycket som fakturerades för det året. Det är åtminstone ett rimligt antagande från det anropet. Lägg till bra namn på paket, procedur och variabler så är det rimligt att man förstår vad som kommer ske, varför och vilka variabler som kan komma att ha uppdaterade värden efteråt.

**Variabler**

Alla variabler skall ha prefix som visar vilken typ de är.

Globala variabler skall bara i undantagsfall användas och de skall aldrig deklareras i paketspecifikationen. Gör de det så kan de ändras från vilket annat paket som helst. Undantaget är förstås konstanter som kan definieras utan risk att bli ändrade från kod utanför paketet.

Variabler skall också ALLTID definieras med hjälp av %TYPE när det finns någon sådan att basera det på.

**Namnsättning**

Följande prefix används:

|  |  |  |
| --- | --- | --- |
| **Prefix** | **Betydelse** | **Exempel** |
| gv | Global Variabel  Dvs., en variabel deklarerad i bodyn av ett paket. | gv\_username |
| lv | Lokal Variabel  Dvs., en variabel deklarerad i en procedure eller funktion. | lv\_cust\_name |
| p | Parameter  Dvs., en variabel som skickas in i funktioner eller proceduren.  Notera att det finns ett andra prefix här, det avgör hur variabeln används. in för in-parametrar, out för out-parametrara och io för IN+OUT-parametrar. | p\_in\_cust\_no |
| sql | Prefix på paket som innehåller SQL. Notera att det även skall ha ett av suffixen ro/rw beroende på vilken sorts SQL det innehåller. Andra paket skall inte innehålla någon SQL. | invoices\_sql\_ro |

Följande suffix används:

|  |  |  |
| --- | --- | --- |
| **Suffix** | **Betydelse** | **Exempel** |
| rw | Suffix på ett paket som uppdaterar data. | inv\_billrun\_rw |
| ro | Suffix på ett paket som inte uppdaterar data. | inv\_billrun\_ro |

Det är värt att märka att i många projekt är kortnamn för tabeller använda för att minska onödigt långa namn på paket, foreign keys och andra namn som baseras på namnet på en eller flera

**Formatering av kod**

All formatering av kod skall göras på ett enhetligt vis. För att se till att den görs på ett enhetligt vis så används SQL Developer version 3 eller högre. För att förenkla inställningen av hur formattering skall göras så bör standardprofilen importeras och användas enligt nedan.

**Beskrivning av kodens formatering**

All kod skrivs i lower case, moderna editorer färglägger koden så bra att shiftläge inte behövs. Lower case ger luftigare och mer lättläst kod.

Nyckelord justeras högerställt, dvs t i select står i samma kolumn som m i from.

Villkor (=,>, < osv) justeras så de är i en jämn kolumn.

Indentering görs med två blanksteg. Tabbar får användas för framstegning vid redigering så länge de inte sparas med koden.

Det skall vara en radbrytning före komma, AND/OR, och ||. Därefter skall de högerjusteras i stil med nyckelorden.

Ingen radbrytning för subselecter, men för parenteser och för join kommandon.

**Notera**

Det finns buggar som gör att den formatering vi vill uppnå inte kan uppnås idag, men den effekt som standardprofilen ger är ändå acceptabel då den gör mycket av jobbet för oss och det viktigaste är att uppnå enhetligt formaterad kod. Då formateringen fungerar ännu bättre i framtiden kommer profilen ändras så att kod därefter formateras som vi vill. Eftersom formateringen kan göras om när som helst utan en massa arbete så är kostnaden att ändra standarden för formatering väldigt låg.

Den stil vi vill åstadkomma i slutändan vore kod som ser ut så här:

select customer\_no

,customer\_name

,invoice\_cycle

,count(\*)

from customer a

inner join customer\_invoice\_cycle b

on a.customer\_no = b.customer\_no

where a.active = ’Y’

and b.invoice\_date = (select max(c.invoice\_date)

from customer\_invoice\_cycle c

where a.customer\_no = c.customer\_no)

group by customer\_no

,customer\_name

,invoice\_cycle

order by count(\*) desc;

Som synes är formateringen hårt styrd och välformaterad kod har ett stort värde i sig. Ju mer komplex SQL, desto viktigare blir det att lätt kunna förstå SQLen genom att bara läsa den.

Som sagt så kan vi inte få exakt ovanstående formatering idag då det finns bristande funktionalitet och buggar, men det visar vad vi eftersträvar och vad vi kommer att försöka uppnå i takt med att formateringsverktyget i SQL Developer förbättras ytterligare. Det mesta fungerar redan, de två större brister vi vet om är att kolumnerna i SELECT inte hamnar indragna som de skall och att AND/OR inte högerställs som ett nyckelord (select/from/where…). Join kommandon i ANSI-syntax hamnar inte heller indragna som nyckelord. Det gör SQLen lite mer svårtläst, men det är småsaker jämfört med vad som fungerar mycket bra i dagens version.

**Manuell inställning**

**Alignment**

|  |  |
| --- | --- |
| **Inställning** | **Val** |
| Right-Align Master Keywords | Vald |
| Align equal Signs | Vald |
| Align Operator Signs | Vald |
| Align on AS Keyword | Vald |
| Align on Commas | Ej Vald |
| Align line comments (--) | Vald |
| Align || at end of line | Ej Vald |
| Align variable declaration for stored procedures | Vald |

**Indentation**

|  |  |
| --- | --- |
| **Inställning** | **Val** |
| Spaces | 2 |
| Use Tabulator | Ej Vald |
| Indent Main Keyword 2x | Ej Vald |
| Indent AND/OR | Vald |

**Line Breaks**

|  |  |
| --- | --- |
| **Inställning** | **Val** |
| Schema Type | large SQL |
| After SELECT, FROM, WHERE | Ej Vald |
| Before Comma | Vald |
| Before AND/OR | Vald |
| Before “||” | Vald |
| After Comma | Ej Vald |
| After AND/OR | Ej Vald |
| After “||” | Ej Vald |
| Number of Commas per line | 1 |
| For brackets in AND/OR | Ej Vald |
| For subselect brackets | Ej Vald |
| For JOIN statements | Vald |
| Max Line Width | 120 |
| Threshold for small SQL | 1 |
| Force line break before line comments | Ej Vald |
| More New Lines | Vald |

**CASE line breaks**

|  |  |
| --- | --- |
| **Inställning** | **Val** |
| CASE linebreaks | Vald |
| CASE | Vald |
| WHEN | Vald |
| THEN | Ej Vald |
| ELSE | Vald |
| AND/OR | Vald |

**White Space**

|  |  |
| --- | --- |
| **Inställning** | **Val** |
| Spaces around Operators | One Space |
| Spaces around Commas | No Spaces |
| Space around Brackets | No Spaces |

**Other**

|  |  |
| --- | --- |
| **Inställning** | **Val** |
| Delete Comment | Ej Vald |
| Put -- comment between /\* ... \*/ | Ej Vald |
| Force formatter output on difference | Vald |
| Case change | Whole SQL lowercase |

**Import av standardprofilen**

Filen code\_style.xml innehåller vår template för formatering av kod. Den implementerar förstås inte all standard, utan bara hur koden skall formateras.

Importera den via properties i SQL Developer och välj sedan Database -> SQL Formatter och välj där import. När importen gjorts kommer ett nytt val i undersektionen Oracle Database finnas. Den heter Kentor SQL och är den inställning som skall användas för all kod som är produktionsklar.

Formatering utförs sedan genom att i ett redigeringsfönster högerklicka och välja format.

**Felhantering**

Att ha en konsekvent felhantering hjälper dels koden att fungera bättre, men också så att fel loggas på ett förväntat sätt och att felkoder inte återanvänds av misstag. Genom att dels logga och dels använda exceptions på samma vis överallt så får vi felrapporter som är tydligare och enklare att jobba med.

Genom att generalisera koden så att inget hårdkodas i andra program än i själva felhanteringsmodulen så kan förbättringar läggas till utan att behöva ändra varje anrop.

**Definiera felkoder**

Varje fel som kan tänkas uppstå dokumenteras på två ställen. Det första är i paketet kentor\_errors där en felkod defineras på följande vis.

invalid\_value\_in\_parameter\_exc exception;

invalid\_value\_in\_parameter\_num constant integer := -20000;

pragma exception\_init(invalid\_value\_in\_parameter\_exc, -20000);

Den första raden skapar en exception, den andra skapa en konstant för felnumret, och den tredje kopplar felnumret till den skapade exceptionen. Notera att felnumret i rad tre måste vara hårdkodat, det är inte tillåtet med en variabel där, inte ens om det är en konstant.

Genom att lägga upp det i en gemensam fil så minskar risken att samma nummer används på flera ställen och genom att definiera ett namn för den så kan felkoderna numreras om senare utan att annan kod behöver ändras.

När ett nytt fel läggs till så ta bara ett lägre än det sista felet i paketet. Notera att det här är ett paket med bara specifikation, det kommer aldrig ha en body.

I vissa fall kan det vara önskvärt att reservera felkoder så att man får en sekvens av dem för ett visst funktionellt område. I det fallet skapar man helt enkelt upp dem i kentor\_errors och benämner dem “reserved\_<område>”. När ett område så är färdigprogrammerat måste man ta bort dem som inte användes och om man nu inte är längst ner i listan (för att någon reserverat ett senare nummer) så måste en MYCKET tydlig kommentar läggas in så de felnummer som nu inte är använda används innan nummer senare i listan bokas. Det är också rimligt att endast i undantagsfall reservera sviter om mer än 10 nummer. Anledningen är att det endast finna 1000 nummer att tillgå och om för många blir reserverade kan de förstås ta slut.

Därefter behöver felkodens betydelse läggas till i tabellen kentor\_error\_text. Tabellen används för att felets betydelse skall kunna fås fram med SQL.

Kom ihåg att den nya raden behöver läggas till i DDL paketet så den kommer med när schemat skapas om. Skicka därför ett email med en gång till den som underhåller DDL/DML.

**Anropa felrutiner**

De felkoder som skapats i kentor\_errors kan nu användas när ett fel uppstår.

Syntax:    
exec kentor\_error\_mgr.report\_and\_**stop** (p\_in\_error\_num => kentor\_errors.invalid\_value\_in\_parameter\_num   
 ,p\_in\_message\_str => 'Some additional information');

Alternativt:

exec kentor\_error\_mgr.report\_and\_**go** (p\_in\_error\_num => kentor\_errors.invalid\_value\_in\_parameter\_num   
 ,p\_in\_message\_str => 'Some additional information');

De två anropen skiljer bara i huruvida man vill avsluta körningen eller bara vill logga felet. Om man vill avsluta så skapas en exception, annars så skippas det. I båda fallen skrivs felet till kentor\_error\_log i en autonomous transaktion så det behålls där även om jobbet i sig senare skulle rulla tillbaka.

Det är alltså vad som behövs för att logga ett fel och få en exception skapad om så önskas. P\_in\_message\_str är till för utökad information i tillägg till den mening som lagts in i kentor\_error\_text. Behövs den inte kan den skippas eller skickas in med null.

En referens till job\_id loggas så att loggningar från olika körningar kan hållas isär.

**Assert**

Det finns också en rutin för att försäkra sig om att något villkor är uppfyllt. Den kommer vanligen användas för ”assert” funktionalitet i procedurer och funktioner där man vill verifiera att de krav som ställs på inparametrer uppfylls. T ex att ett tvåsiffrigt nummer som representerar en månad inte innehåller ett värde som överstiger 12.

Syntax:   
 kentor\_error\_mgr.assert  
 (p\_in\_condition\_bool => v\_month\_number <= 12

,p\_in\_message\_str => 'Month validation'

,p\_in\_exception\_num => kentor\_errors.invalid\_value\_in\_parameter\_num);

P\_in\_condition\_bool tar ett boolskt värde såsom resultatet av ett villkor. P\_in\_message\_str är bara en text som skrivs ut till dbms\_output och skickas vidare till report\_and\_stop.

P\_in\_exception\_num är en text sträng som anger vilken exception som skall skapas ifall p\_in\_condition\_bool resulterar i ett falskt värde.

Assert kan användas dels i assert delen av koden, men också i andra delar där det passar bättre med ett anrop på det här viset än med att skriva olika if satser för att verifiera olika möjliga felscenarier.

**Tolka data i kentor\_error\_log**

Anropen till report\_and\_\* resulterar primärt i att rader loggas i kentor\_error\_log. Tabellen är till stor del självförklarande, här följer en enkel beskrivning av vad man skall vänta sig att hitta i den.

Varje rad motsvarar ett fel som koden instrumenterats för att logga. Det skall representera en felsituation som är möjlig men inte förväntad. Raderna här är inte en körningslog för att se hur körningen fortskred, för det används i stället instrumenteringen som är dokumenterad på annan plats i det här dokumentet. Rader i tabellen visar på problem som behöver åtgärdas och vanligen behöver programmet sedan köras om.

Varje rad innehåller följande kolumner:

* Aud\_job\_no Unikt ID för varje körning. ID:t sätts vid initieringen av instrumenteringen för jobbet. Aud\_job\_id kan kopplas till raden för instrumenteringen i kentor\_aud\_jobs (förutsatt att jobbet är instrumenterat) där mycket information om jobbet kan hittas såsom dess namn, vem som körde det, om en tracefil skapades osv.
* Error\_num En referens till det fel som uppstod. Felets betydelse kan slås upp genom att koppla det till en rad i kentor\_error\_text. Felet är också dokumenterat i paketet kentor\_errors som innehåller det variabler som används för att hantera felet.
* Error\_str Medskickad text från det program som loggade felet. Dvs., det innehåller inte beskrivningen från error\_text, utan bara eventuell kompletterande information.
* Log\_ts Timestamp för databas serverns tid vid tillfället då felet loggades.
* Call\_stack Komplett call stack för de procedurer som anropats för att komma fram till loggningen. Det sista är förstås felhanteringens egen. Den första dessförinnan är ”vanligen” den plats där felet identifierades och anrop till felhanteringen gjordes.

**Instrumentering**

Instrumentering av kod är kritiskt för varje applikation som skall ha någon form av prestandakrav på sig förr eller senare. Utan ordenlig instrumentering blir de flesta applikationer svarta lådor i vilket det är mycket svårt att analysera varför något plötsligt tar lång tid. För att underlätta instrumenteringen av batchkod har vi standardiserat hur det görs och vad vi sparar. Det gör att alla jobb som körs kan analyseras på samma vis och med samma precision per del som instrumenterats. Planen är att instrumentera all kod på mycket detaljerad nivå så varje del som kan ta mer än ett par sekunder kan ses i en instrumenteringslog och analys om var tid faktiskt spenderas kan utföras.

Detta för batchjob, online bör istället använda den instrumentering som en utvecklingsmiljö idag skall tillhandahålla. Om inte kan en liknande modell byggas för online-delarna.

**Initiering.**

Initieringen kan bara köras en gång per session och vad det gör är att etablera ett unikt ID för en körning, liksom att spara viktig information om vad jobbet är och hur det startades.

Initieringen görs genom att ha en sessionsinitiering i varje paket jobb-paket. Det är det som anropas för att starta ett jobb, vilket gör att det första som körs i sessionen är det paketets sessionsinitiering. Den måste ligga sist i paketet så det ser ut ungefär så här.

create or replace package body my\_pkg as

procedure proc\_a(p\_in\_custno in mumber) is

begin

null;

end;

...

/\* Package Initialization code \*/

**begin**

**kentor\_instrument.initialize(p\_in\_batch\_id => 1)**

end my\_pkg;

Den parameter som skickas in anger vilken batch det är som körs. De som kan köras måste först definieras och det gör man i tabellen kentor\_batch\_parms.

ID:t måste vara unikt och sätts bara till ett högre än det högsta som redan finns definierat. Namnet är ett funktionellt namn på batchen. Indikatorn för trace anger ifall en SQL trace skall skapas. Är den satt till Y så skapas en tracefil som innehåller både bindvariabler och wait information. Den skall vara N utom då man behöver en trace fil. Det finns också en kolumn för vilken instrumenteringsnivå man vill köra på. Den är satt 1-4, där ett är full real-time loggning, 2 är full loggning men spara endast efteråt, 3 är att inte spara till tabell, och 4 är att inte köra med instrumentering alls.

Instrumenteringsnivå påverkar inte initieringen alls, utan bara hur mätning av instrumenterade aktiviteter hanteras.

När initialiseringen körs så skapas en rad i tabellen kentor\_aud\_jobs som visar varje körning av ett instrumenterat jobb. Den tabellen innehåller en rad med kolumner från USERENV kontextet i databasen (unikt per session). Exempel på information är vilken databas användare som kör jobbet liksom vilken operativsystemsanvändare som loggade på. Dessutom sparas om en tracefil skapades för jobbet. Den tracefilen är en sql trace med bind variabler och wait information. Job\_id är unikt för körningen, medan batch\_id är en referens till kentor\_batch\_parms där olika batchjob definieras så att man kan avgöra vilken sorts jobb det var som olika körningar tillhör.

Samtidigt så sätts instrumentering i databasen via dbms\_application\_info. Module sätts till namnet på batchen som tas ifrån kentor\_batch\_parms. Därigenom kan man starta en trace för alla körningar av en viss batch. Action och client\_info sätts till INSTRUMENT.

**Starta en aktivitet**

En aktivitet är en del i ett program som kan ta någon tid. Tar något mer än ett par sekunder i värsta fall så bör det instrumenteras. Det betyder att de flesta procedurer och funktioner skall ha åtminstone en start i början och en stop i slutet. I många fall görs mer än en sak som kan ta tid i varje begin-end block. Även de bör instrumenteras med start och stop anrop.

Flera anrop i rad till start resulterar i konkatenering av dem så att det första innehåller mätningen av alla steg tills det första avslutas medan det sista bara mäter tiden just det steget tar.

För att starta en aktivitet så anropas start\_time i paketet kentor\_instrument. Anropet ser ut så här:  
kentor\_instrument.set\_start\_time(p\_in\_function\_op => ’AAA’)

AAA kan i det här fallet betyda att funktionaliteten för ”read invoice” startas. Koden kommer inte ha en direkt koppling till betydelsen. Översättningen mellan kod och betydelse görs genom att koppla koden mot tabellen kentor\_activities. Genom att hålla funktionskoderna till 3 tecken så kan 16 nivåer av anropade funktioner loggas.

AAA är hårdkodat i anropet ovan. Det skall det förstås inte vara, utan det skall skapas en konsonant i paket kentor\_activity\_codes.

När funktionen anropas så sker två saker. Dels sätts client\_info till vad det var förut och så läggs ett kolon till och därefter det funktionsnamn man anropade med. På så vis blir kod som anropas av en funktion inkluderat i om man mäter på vad som sker i den funktionen. Dvs., om man mäter vad som sker i AAA:AAB så får man också veta vad som sker i AAA:AAB:AAC och AAA:AAB:AAC:AAD osv. En rad sparas i tabellen kentor\_aud\_jobs\_detail (om man ställt in instrumenteringsnivån för jobbet för det) så att man senare kan se när olika steg startade och stoppades. Även här används samma vis att bygga upp funktionsnamnen så att yttre funktioner inkluderar de anrop som sker.

Raden i kentor\_aud\_jobs\_detail skippas om log\_table\_ind är satt till N. Det är för att funktioner som anropas tätt i en loop inte skall ta upp en stor del av den totala tiden. Den används mest i anrop för fetch och insert så att man kan få det instrumenterat i databasen utan att logga tusentals med rader utom i test då man kan vilja analysera hur lång tid varje fetch tar och om varje fetch tar lika lång tid eller om det finns stora skillnader. Ett alternativ är att ställa in att spara instrumenteringen först efteråt så att man kan få med detaljerna utan att betala den höga kostnaden för enstaka insert och update kommandon.

**Stoppa en aktivitet**

På samma vis som en aktivitet startas så skall koden instrumenteras så att den stoppas när den är avslutad. Det görs genom att anropa stop\_time i paketet kentor\_instrument. Anropet ser ut så här:   
kentor\_instrument.set\_stop\_time(p\_in\_function\_op => ’AAA’  
   ,p\_in\_rows\_num => 1234)

P\_in\_rows\_num är frivillig och skall bara skickas med då aktiviteten bearbetade rader i tabell eller kollektion som då bör rapporteras.

AAA är hårdkodat i anropet ovan. Det skall det förstås inte vara, utan det skall skapas en konsonant i paket kentor\_activity\_codes.

När det här anropet sker kollas det att den senast startade funktionen är AAA eftersom aktiviteter måste stoppas i samma ordning som de startas.

Därefter sätts client\_info till vad det blir om man tar bort det funktionsnamn som skall stoppas. Dvs., om client\_info är satt till AAA:AAB:AAC och ett anrop sker för att stoppa AAC (det enda acceptabla anropet) så blir client\_info därefter satt till AAA:AAB.

På samma vis uppdateras raden för den aktiviteten i kentor\_aud\_jobs\_detail så att en stoptid sätts. Det gör att vi senare kan aggregera och utvärdera hur lång tid olika aktivteter tar.

**Mappning av aktivitetsnamn**

För att vi skall kunna ha korta aktivitetsnamn i instrumenteringen och samtidigt kunna skriva ut rapporter med funktionellt beskrivande namn behöver vi registrera varje aktivitetskod. Det låter oss också kontrollera vilka som skall loggas till en tabell och vilka som inte skall det.

Registrering görs i tabellen kentor\_activities. Nyckel är function\_op\_str som kopplar mot kentor\_aud\_jobs\_detail. Därtill finns ett attribut activity\_name\_str i vilket ett funktionellt namn för aktiviteten skall läggas in. Notera att det skall hållas teknik och terminologifritt så att verksamhetsfolk och chefer kan förstå ungefär vad det betyder. Det finns också ett attribut proc\_name\_str som visar i vilken procedur/funktion som aktivitetskoden används.

Kom ihåg att alla rader som läggs till måste läggas in i DDL-scriptet så att raderna skapas när schemat skapas om.

**Rapportering**

Denna SQL ger en hyfsat prydlig rapport. Den rapporterar på alla jobb som körts så ytterligare begränsning behövs.

select substr(lpad(c.activity\_name, length(c.activity\_name) + (3 \* level\_num) + 1, ' '),1 ,80) Activity\_name\_str

,substr(elapsed\_tm, instr(elapsed\_tm, ':') + 1, instr(elapsed\_tm, '.') + 1 - instr(elapsed\_tm, ':')) "TIME (MIN.SEC.MS)"

,rows\_num

from (

select aud\_job\_no

,substr(function\_op\_str, instr(function\_op\_str, ':', -1) + 1) activity\_id

,length(function\_op\_str) - length(replace(function\_op\_str, ':', '')) level\_num

,stop\_ts - start\_ts elapsed\_tm

,rows\_num

from kentor\_aud\_jobs\_detail

) a

inner join kentor\_aud\_jobs b

on a.aud\_job\_no = b.aud\_job\_no

inner join kentor\_activities c

on a.activity\_id = c.activity\_id

;

**Job wrapping**

Det här är ett koncept som ofta används för att se till att viss initiering görs innan funktionell kod startas och det tvingar också ett system att använda ett konsekvent api för de funktioner som omfattas.

För batch används det för varje jobb så att instrumentering startas och initieras innan jobbet börjar.

**API**

För att kunna låta alla jobb ha olika signaturer så implementeras job-wrappingen via paketinitiering. Dvs, det är inte en wrapping i egentlig mening utan det blir lite duplicerad kod som körs första gången ett paket körs i en viss session. Den koden kommer dock bara anropa en funktion så det mesta av vad som görs göms i infrastrukturen.

När ett nytt jobb skapas skall proc\_name\_str läggas till i kentor\_batch\_parms tillsammans med annan data som behövs där.

I paket som kör jobb skall följande kod ligga sist i paketet. Det är alltså sessionsinitieringskod som vi utnyttjar för att få jobb initieringen gjord för instrumentering.

-- Package initialization for the session

begin

kentor\_instrument.initialize(p\_in\_batch\_id => 0);

0 är Id för jobbet enligt kentor\_batch\_parms.

Dessutom måste proceduren main (som skall vara jobbets enda exponerade procedur) avslutas med följande kod för att se till att data för instrumenteringen skrivs till databasen. Den lagras bara i en collection under körningen och måste därför sparas när jobbet är klart.

**Att programmera med infrastrukturen**

I tillägg till ovanstående beskrivning och förklaring till hur de olika delarna fungerar och hänger ihop så kan nog en enkel beskrivning över ett föreslaget arbetssätt förtydliga hur man arbetar med infrastukturen.

1 Kopiera följande filer till nya namn.

kentor\_ref\_job.pkb

kentor\_ref\_job.pks

tst\_customer\_sql.pkb

tst\_customer\_sql.pks

Ändra namnet till ett funktionellt beskrivande för vad paketets kod gör.

De olika filsluten visar vad filerna kommer innehålla. Pks är filslutet för ett pakets specifikation, pkb är ett pakets body. Proc som filslut visar att det är en procedur och inte ett paket.

2 Lägg upp batchen

Skapa en ny rad i tabellen kentor\_batch\_parms för det jobb du håller på att skapa. Den tabellen innehåller följande kolumner.

* BATCH\_ID Unikt nummer för batchen. Tag första lediga nummer.
* BATCH\_NAME Funktionell beskrivning för batchen. Hamnar i instrumentering och trapporter.
* TRACE\_IND Avgör ifall en SQL trace skall startas. Skall sättas till N och bara ändras till Y vid tillfällen då en trace behövs.
* PROC\_NAME\_STR Namnet på den procedur som innehåller aktiviteten. I referensimplementationen är det **kentor\_ref\_job.main.**
* INSTRUMENTATION\_LEVEL\_NUM Vilken sorts instrumentering av de aktiviteter som är instrumenterade önskas. Följande val är möjliga. 1 – FULL – Spara instrumenteringsdata i tabell och gör det löpande medan jobbet körs. 2 – DELAY – Spara instrumenteringsdata i tabell, men skriv det inte förrän jobbet avslutas. 3 – NOTABLE – Skriv inte instrumentering till tabell. 4 – NONE – Ingen instrumentering alls, inte ens via DBMS\_APPLICATION\_INFO.  Sätts till 2 i alla fall och ändras sedan bara för att passa behov under utveckling. Anledningen är att FULL tar mycket resurser med så detaljerad instrumentering som är önskad, medan 3,4 resulterar i ingen instrumenterad körlogg. DELAY däremot ger en full logg, men den sparas endast då jobbet avslutas.

När raden är skapad måste samma information också skickas till den som underhåller DDL och referensdata så att den skapas om när lösningen flyttar till nya miljöer (dev-test-prod o dyl).

3 Skapa mätpunkter (aktiviteter)

Aktiviteter behöver skapas, och vanligen blir det ganska många. Gissningsvis skulle 10-20 skapas även om jobbet är mycket enkelt. Alla operationr som an tänkas vara av intresse för utvärdering någon gång skall ha en aktivitet. De läggs upp i tabellen kentor\_activities som har följande kolumner.

* ACTIVITY\_ID Unik nyckel för en aktivitet. Värdet har formatet XXX där det lägsta är AAA och det högsta är ZZZ. Tag nästa värde i serien för nya aktiviteter.
* ACTIVITY\_NAME Funktionellt namn för aktiviteten. Skall vara lämpligt för rapporter till chefer och andra icke tekniska användare.
* PROC\_NAME\_STR I vilken procedur eller paket finns aktiviteten. Kvalificerat med paketnamn.
* LOG\_TABLE\_IND Skall aktiviteten loggas till tabell? Sätts till Y om inte speciell omständighet råder.

Kom ihåg att skicka ett email om de nya aktiviteterna till den som ansvarar för DDL och referensdata.

4. Skapa felkoder

Felhantering som hanterar det som kan uppkomma men inte bör måste först registreras i kentor\_errors och dessutom läggas till i kentor\_error\_text. All felhantering bör göras med exceptions och då använda det paket och den tabell vi använder för det. Anrop för felhantering och loggning skall sedan göras till kentor\_error\_mgr som tidigare beskrivet.

Tabellen kentor\_error\_text innehåller följande kolumner.

* ERROR\_ID Den felkod som valts för felet. Sätt det till ett lägre än det lägsta som redan är definierat.
* ERROR\_TEXT Beskrivande text för felet. Så specifikt som möjligt utan att för den skull göra felet omöjligt att återanvända på annat ställe för liknande fel. Vid felanrop kan ytterligare specifikation skickas med. Det mer specifika loggas då till kentor\_error\_log, medan det generella inte lagras igen utan kan ses här genom att koppla från kentor\_error\_log till kentor\_error\_text.

Kom ihåg att skicka ett email om de nya aktiviteterna till den som ansvarar för DDL och referensdata.

5. Skriv jobbet

Dags att programmera den kod som paketet skall ha.

6. Övrig verifikation

Innan det nya jobbet är helt klart bör förstås lite till valideras.

En koll på att instrumenteringen blev som tänkt och att de nya aktivitetskoderna används skall göras. Instrumenteringen lagras i kentor\_aud\_jobs\_detail.

Verifiering av att de använda felkoderna och dess rapportering fungerar är också nödvändig. Den informationen lagras i kentor\_error\_log.