Пример ВКР «SPbPU-student-thesis-template». Версия от 25 мая 2021 г.

Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий

Работа допущена к защите
Руководитель ОП М
_____ В.Г. Пак
«_____» _____ 2021 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА
РАБОТА БАКАЛАВРА
РАЗРАБОТКА ПАКЕТА ДЛЯ СТАНДАРТИЗАЦИИ И АВТОМАТИЗАЦИИ ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ В PL/SQL

по направлению подготовки 02.03.03 Математическое обеспечение и администрирование информационных систем
Направленность (профиль) 02.03.03_YY Наименование направленности (профиля) образовательной программы

Выполнил
студент гр. 3530203/70102 М.Н. Батищев

Руководитель
доцент,
к.т.н, звание О.Ю. Сабинин

Консультант
должность, степень О.Ю. Сабинин

Консультант
по нормоконтролю В.А. Пархоменко

Санкт-Петербург
2021

САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО

Институт компьютерных наук и технологий

УТВЕРЖДАЮ
Руководитель ОП М
_____ В.Г. Пак
«_____» _____ 2021г.

ЗАДАНИЕ
на выполнение выпускной квалификационной работы

студенту Батищеву Михаилу Николаевичу гр. 3530203/70102

1. Тема работы: Разработка пакета для стандартизации и автоматизации обработки исключительных ситуаций в PL/SQL.
2. Срок сдачи студентом законченной работы: 19.05.2021.
3. Исходные данные по работе: документация Oracle Database [3.2], схема HR для тестирования работы из скрипта [3.4], основной литературой является руководство [3.1], а также статья [3.3].
   3.1. *Фейерштейн С. П. Б.* Oracle PL/SQL. Для профессионалов. — 6-е изд. — СПБ: Питер, 2015. — 1024 с.
   3.2. Database Administrator's Guide. — URL: https://docs.oracle.com/cd/E11882_01/server.112/e25494/toc.htm.
   3.3. Handling PL/SQL Errors. — URL: https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm#LNPLS007.
   3.4. Source code for HR scheme. — URL: https://download.oracle.com/oll/tutorials/DBXETutorial/html/module2/les02_load_data_sql.htm.
4. Содержание работы (перечень подлежащих разработке вопросов):
   4.1. Выявление существующих проблем при обработке исключительных ситуация стандартными способами.
   4.2. Анализ возможных способов устранения выявленных недостатков.
   4.3. Анализ существующих пакетов для обработки ошибок в Oracle.

4.4. Исследование и разработка методик автоматизации и стандартизации работы с исключительными ситуациями.
4.5. Разработка программы на PL/SQL для автоматизации и стандартизации обработки ошибок.
4.6. Аппробация разработанного пакета.

5. Перечень графического материала (с указанием обязательных чертежей):
   5.1. Диаграмма архитектуры разрабатываемого пакета.
   5.2. ER модель используемой для тестирования схемы HR.
6. Консультанты по работе:
   6.1. Ассистент ВШИСиСТ, В.А. Пархоменко (нормоконтроль).
7. Дата выдачи задания: 26.02.2021.

Руководитель ВКР _____ О.Ю. Сабинин

Задание принял к исполнению 26.02.2021

Студент _____ М.Н. Батищев

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Наша жизнь прочно связана с цифровыми технологиями. Сегодня трудно представить наше существование без компьютера или интернета. Большинство повседневных действий переводится в онлайн. С каждым годом все больше появляется новых информационных систем, а любая информационная система, будь то приложение или вебсайт, тем или иным способом взаимодействует с данными, которые нужно правильно хранить и обрабатывать. Для взаимодействия с данными большого объема были разработаны базы данных.

Ни для кого не является секретом, что одним из важных этапов разработки любого программного обеспечения является обработка исключительных ситуаций. Такие ситуации не возникают в идеальных условиях, но повсеместно встречаются в нашей жизни. Мы не можем предусмотреть абсолютно все возможные варианты развития событий, но постараться предугадать самые часто встречающиеся ошибки, нам никто не запрещает.

При разработке программного продукта любой программист задается вопросами разного рода. Что будет с машиной, если неожиданно перестанет поступать питание? А если пользователь выйдет из приложения, не сохранив отчет? Что нужно передать клиенту, если запрашиваемых данных не существует? Продолжать список можно бесконечно, но если мы предполагаем, что такая ситуация может возникнуть, то не лишним будет попробовать защититься от последствий. При некоторых исключительных ситуациях такие последствия могут быть фатальными, стоимость необработанной ошибки в крупных кампаниях может исчисляться миллионами долларов, а для некоторых сфер, может исчисляться и в человеческих жизнях. Поэтому для разработчиков нужно предоставить максимально удобный и простой способ для обработки исключительных ситуаций.

Одной из самых популярных систем управления базами данных уже достаточно долгое время является Oracle Database. Она славиться своей надежностью, производительностью, масштабируемостью и безопасностью.

Oracle предоставляет разработчикам баз данных очень гибкий и мощный механизм для работы с ошибками, но он не лишен недостатков, которые могут создать существенные проблемы для групп разработчиков, которые хотят построить систему управления ошибками, обладающую свойствами надежности, содержательности и последовательности.

Целью данной работы является разработка пакета, нацеленного на модернизацию и улучшение существуеющего в Oracle механизма работы с исключительными ситуациями.

Для достижения поставленной цели необходимо:

A. Реализовать пакет на языке PL/SQL, предназначенный для улучшения взаимодействия с существующей системой обработки исключительных ситуаций
B. Протестировать работу пакета
C. Сравнить разработанное решение со стандартным способом работы с ошибками

ГЛАВА 1. МЕХАНИЗМЫ ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ В ЯЗЫКЕ PL/SQL

В данной главе рассматривается базовые концепции работы с исключительными ситуациями в языке Oracle PL/SQL.

В параграфе 1.1 привидена основная информация об ошибках в PL/SQL. В параграфе 1.2 описаны методы работы с именованными исключениями. Возможности для инициирования исключительных ситуаций описаны в параграфе 1.3. Про обработку ошибок рассказывается в параграфе 1.4.

1.1. Основные концепции обработки исключительных ситуаций в PL/SQL

Исключительными ситуациями в языке PL/SQL считаются любые ситуации, которые не должны возникать при нормальном выполнении программы[3].

Под данное определение попадает достаточно большое множество ситуаций, в качестве примера исключений можно привести следующие события: отсутствие данных, ошибки работы экземпляра, непредусмотренные программой действия пользователей. Любая из приведенных выше ситуаций может привести к нежелательному результату, если не будет вовремя найдена и обработана.

В языке PL/SQL ошибки перехватываются и обрабатываются в отдельном месте в коде, называемым блок обработки исключений. Такой подход позволяет работать с ошибками как с событиями, передавая управление нужному блоку сразу после возникновения исключительной ситуации, вне зависимости от того в какой конкретно месте возникла данная ошибка.

Продемонстрируем преимущества такого метода. На рис.1.1, показан вариант линейной обработки ошибок, в таком случае, после каждой команды SELECT мы должны проверить наличие данных.

```
BEGIN
    SELECT ...
        -- Проверка ошибки 'no data found'
    SELECT ...
        -- Проверка ошибки 'no data found'
    SELECT ...
        -- Проверка ошибки 'no data found'
```

Рис.1.1. Пример кода с последовательной обработкой ошибок

Если нам заранее известно, что имеется необходимость во всех данных из каждой команды SELECT, а при отсутствии хотя бы одних из них, мы должны

выполнить какие-либо действия, то имеется возможность сильно упростить код, как показано на рис.1.2.

```
BEGIN
    SELECT ...
    SELECT ...
    SELECT ...

    ...

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        -- Обработка ошибок 'no data found'
```

Рис.1.2. Пример обработки ошибок в отдельном блоке

В таком случае обработка всех ошибок данного типа будет производиться в одном месте, отлаживать и дорабатывать такую программу будет гораздо проще, по сравнению с первым случаем. Также вынесение обработчика в отдельный блок, позволяет отделить логику работы основной программы, от раздела исключений, что позволяет уменьшить количество кода и повысить его читаемость.

В PL/SQL исключения бывают двух типов: системные и пользовательские. Системные исключения – это исключения, которые инициируются ядром PL/SQL в случае нарушением программы правил исполнения установленных Oracle. Пользовательские исключения, в свою очередь, определяются программистом и обычно связаны с конкретным приложением.

Каждая ошибка имеет свой номер, но обрабатывать исключения можно только по их имени. В Oracle имеется несколько предопределенных ошибок, это часто возникающие ошибки, для которых заданно имя в пакете STANDARD.

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
CASE_NOT_FOUND	ORA-06592	-6592
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Рис.1.3. Предопределенные ошибки из пакета STANDARD

На рис.1.3, приведены примеры предопределенных исключений.

1.2. Определение исключений

В подавляющем большинстве приложений определенных Oracle исключений недостаточно, поэтому необходимо уметь правильно определять и работать с пользовательскими ошибками, которые описывают исключительные ситуации для конкретного приложения.

1.2.1. Объявление пользовательских исключений

Пользовательские исключения позволяют программисту определить новые виды ошибок, для ситуаций, которые не покрываются системными исключениями.

Очень важной особенностью пользовательских исключений является то, что взаимодейтсвие с ними не отличается от работы с системными ошибками.

Определение именованных исключений похоже на объявление обычных переменных. Сначала указывается название ошибки, за которым следует ключевое слово EXCEPTION.

```
DECLARE
        past_due EXCEPTION;
        aact_num NUMBER;
BEGIN
        DECLARE
                past_due EXCEPTION;
                aact_num NUMBER;
        BEGIN
                ...
                IF ... THEN
                        RAISE past_due; -- Не обработается
                END IF;
        END;

EXCEPTION
        WHEN past_due THEN -- Не обработает вызванное исключение
                ...
END;
```

Рис.1.4. Пример объявление именованного исключения

На рис.1.4 показан пример объявления именованного исключения под названием past_due. Во вложенном блоке объявляется еще одно исключение с таким же именем, так как вложенный блок не имеет обработчика, то его (локальное) исключение past_due не будет обработано. Внешний блок ничего не знает об исключение past_due из вложенного блока, поэтому его обработчик не сможет отловить и обработать данное исключение[6].

1.2.2. Привязка исключений к кодам ошибок

При помощи директив компилятора можно связать именованное исключение с кодом ошибки. Делается это при помощи команды EXCEPTION_INIT, в которую передается имя исключения, объявленного ранее, и код ошибки. После такой привязки обращаться к ошибке в разделе WHEN можно по имени.

Обычно данная директива используется в двух случаях: для задания имени системному исключению, для которого не предусмотренно предопределенного имени, и для работы со специфичными для приложения ошибками.

```
DECLARE
        deadlock_detected EXCEPTION;
        PRAGME EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
        ... -- Код вызывающий ошибку ORA-00060
EXCEPTION
        WHEN deadlock_detected THEN
                -- Обработка ошибки
END;
```

Рис.1.5. Пример использования директивы компилятора EXCEPTION_INIT

В примере показанном на рис.1.5 происходит привязка ошибки с номером -60 к именованному исключению deadlock_detected.

Хорошим тоном считается вынесение часто используемых ошибок в один пакет и привязка их к номерам, как, например, это сделано в пакете STANDARD со стандартными исключениями. В таком случае, в остальном коде приложения, можно будет обращаться к ошибкам из данного пакета. Следуя такой практике можно повысить читаемость кода за счет стандартизации в определении ошибок.

1.3. Инициирование исключительных ситуаций

Существует несколько способов инициировать исключение. В первую очередь, исключения инициируются Oracle в случае возникновения ошибки. Программист может сам инициировать исключения либо при помощи команды RAISE, либо используя процедуру RAISE_APPLICATION_ERROR. Рассмотрим последние два способа подробнее.

Команда RAISE останавливает нормальное выполнение программы PL/SQL и передает управление обработчику ошибок. Данная инструкция имеет следующий синтаксис.

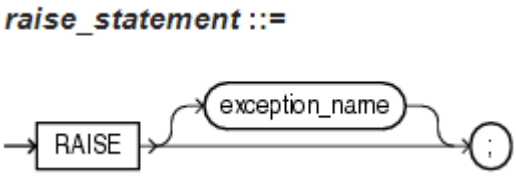

Рис.1.6. Синтаксис команды RAISE

Как видно на рис.1.6 сначала указывается ключевое слово RAISE за которым следует необязательное имя исключения.

Использование данной команды без указания имени исключения, можно только в разделе WHEN обработчика исключений. В таком случае обрабатываемое в данный момент исключение инициируется повторно, а процесс обработки исключения переходит в родительский блок. Это может быть полезно в случаях, когда нам нужно освободить какие-то ресурсы, либо сохранить какие-либо данные, но что делать с возникшей ошибкой в данный момент времени мы не знаем.

```
DECLARE
        out_of_stock EXCEPTION;
        number_on_hand NUMBER := 0;
BEGIN
        IF number_on_hand < 1 THEN
                RAISE out_of_stock; -- Вызываем объявленное исключение
        END IF;
EXCEPTION
        WHEN out_of_stock THEN
                -- Обработка ошибки
                DBMS_OUTPUT.PUT_LINE('Encountered out-of-stock error.');
END;
```

Рис.1.7. Пример использования команды RAISE

На рис.1.7 представлен пример инициирования ошибки с использованием инструкции RAISE. В блоке происходит объявление исключения out_of_stock и переменной number_of_hand проинициализированной значением 0, после чего в условном блоке происходит проверка значения переменной, если оно меньше 1, то происходит инициирование объявленного ранее исключения, после чего исполнение передается блоку обработки ошибок.

Другим способом инициировать программисту исключение является процедура RAISE_APPLICATION_ERROR из пакета DBMS_STANDARD.

```
PROCEDURE RAISE_APPLICATION_ERROR (
   num binary_integer,
   msg varchar2,
   keeperrorstack boolean default FALSE);
```

Рис.1.8. Заголовок процедуры RAISE_APPLICATION_ERROR

Как видно на рис.1.8 данная процедура может принимать три параметра, два обязательных и один необязательный. Первый параметр – это код ошибки, которую необходимо инициировать, в диапазоне от -20999 до -20000. Второй параметр – это сообщение длиной до 2048 байт, которое будет связано с ошибкой. Третий параметр keeperrorstack указывает, нужно ли добавить ошибку к уже имеющимся в стеке (данное поведение используется при указании значения TRUE), или необходим заменить существующую ошибку (значение FALSE, используемое по умолчанию).

```
IF l_customer_credit > l_credit_limit THEN
        raise_application_error(-20111, 'Credit limit exceeded!');
END IF;
```

Рис.1.9. Пример использования процедуры RAISE_APPLICATION_ERROR

Пример на рис.1.9 демонстрирует использование рассматриваемой процедуры. При нарушении бизнес-логики происходит инициирование ошибки с кодом -20111 с информацией об ошибке.

1.4. Обработка ошибок

При возникновении исключительной ситуации нормальное выполнение PL/SQL кода останавливается, а управление получает раздел обработки исключений, если обработчик данного блока не может справиться с такой ситуацией, то управление переходит в родительский блок.

Синтаксис блока обработки исключений имеет следующий вид, показанный на рисунке рис.1.10.

```
EXCEPTION
        WHEN exception1 THEN -- Обработчик exception1
                ... -- Код обработки ошибки
        WHEN exception2 THEN -- Обработчик exception2
                ... -- Код обработки ошибки
        ...
        WHEN OTHERS THEN -- Обработчик для всех остальных ошибок
                ...
END;
```

Рис.1.10. Синтаксис обработчика ошибок

Он может находиться только в конце PL/SQL блока либо вовсе может отсутствовать. Если имя возникшего исключения совпадает с именем, указанным в разделе WHEN, то выполняются команды, находящиеся после ключевого слова THEN. В случае, когда ни один раздел с указанным именем не подходит управление переходит в раздел WHEN OTHERS, если он указан.

Стоит отметить, что ошибка может быть обработана только одним из разделов. После выполнения команд управление передается в вызывающий блок.

1.5. Получение дополнительной информации об ошибке

Периодически, возникают ситуации, когда необходимо узнать дополнительную информацию об возникшей ошибке. В Oracle не существует возможности расширить объект EXCEPTION и записать туда свою информацию. В базе данных существует несколько функций, которые позволяют получить различную информацию из ошибки. К ним относятся: SQLCODE, SQLERRM, DBMS_UNTILITY.FORMAT_ERROR_BACKTRACE и DBMS_UNTILITY.FORMAT_ERROR_STACK. Рассмотрим каждую из них подробнее.

Функция SQLCODE позволяет получить код возникшей ошибки, если вызвана в блоке обработки исключительных ситуаций, в том случае, если данная функция будет вызвана вне обработчика, то будет возвращено значение 0. Для ошибок, определенных пользователя, возвращается значение +1 или, если функции был присвоен код, при помощи директивы EXCRPTION_INIT, то данный код и будет возвращен. При помощи данной функции можно обрабатывать исключения, которым не присвоены имена, по коду. Пример рис.1.11 показывает, как это можно сделать.

```
BEGIN
    ... -- Некоторый код

    EXCEPTION
        WHEN OTHERS THEN
            CASE SQLCODE
            WHEN -1015 THEN
                ... -- Обработка ошибки с кодом -1015
            WHEN -2291 THEN
                ... -- Обработка ошибки с кодом -2291
            END CASE;

            RAISE;
END;
```

Рис.1.11. Обработка ошибок с использованием функции SQLCODE

В данном примере, в обработчике ошибок происходит обработка всех ошибок. В случае заданных кодов выполняется особый код только для их обработки, после чего, при помощи команды RAISE снова возбуждается текущее исключение и передается в родительский код. Данную задачу можно было бы решить и другим способом, например, связать коды, для которых нужна особая обработка, с именами исключений, и указывать в конструкции WHEN их имена. Такой подход является более предпочтительным, но приведенный пример тоже имеет право на существование.

Функция SQLERRM возвращает сообщение ошибки, связанное с данным кодом. Может принимать в качестве параметра код ошибки. Если вызвана без параметров и в обработчике ошибок, то вернет ассоциированное с текущей ошибкой сообщение.

Функция FORMAT_ERROR_STACK возвращает более подробную информацию об ошибке, в отличии от SQLERRM, помимо сообщения в вывод также включен текущий стектрейс ошибки. Стектрейс - последовательность вызовов и состояние окружения в некоторой точке программы. Преимуществом данной функции относительно SQLERRM является то, что SQLERRM строже ограничен в размерах сообщения.

Функция FORMAT_ERROR_BACKTRACE позволяет получить стек вызова до точки, где была получена ошибка. Эта информация может потребоваться для нахождения причин ошибки. Например, после обработки ошибок, мы записываем в специальную таблицу полученный стек вызова. В дальнейшем, когда, нам будет необходимо исправить эту ошибку, мы можем обратиться к данной таблице и получить всю необходимую информацию о месте ее возникновения.

1.6. Логирование информации об ошибке

Большинство ошибок не может быть исправлено сразу, и не все ошибки требуют моментального исправления. Некоторые будут более приоритетными, некоторые менее, одни ошибки могут возникать очень часто, другие же встречаться крайне редко. Нужно правильно сохранять информацию об ошибках, когда придет время для ее исправления, мы должны иметь возможность получить максимум информации. Oracle предоставляет разработчикам пакет DBMS_ERRLOG, в котором содержится одна процедура CREATE_ERROR_LOG. Данная процедура предназначена для логирования ошибок в DML операциях.

```
DBMS_ERRLOG.CREATE_ERROR_LOG (
   dml_table_name              IN VARCHAR2,
   err_log_table_name          IN VARCHAR2 := NULL,
   err_log_table_owner         IN VARCHAR2 := NULL,
   err_log_table_space         IN VARCHAR2 := NULL,
   skip_unsupported            IN BOOLEAN := FALSE);
```

Рис.1.12. Сигнатура процедуры DBMS_ERRLOG.CREATE_ERROR_LOG

На рисунке рис.1.12 представлена сигнатура данной процедуры. Процедура в качестве параметра принимает название таблицы, для которой производилась DML операция, информацию о таблице, в которую нужно занести информацию

(таблица логирования) и флаг skip_unsupported, который позволяет пропустить неподдерживаемые столбцы. К таким столбцам относятся колонки с типами данных LONG, CLOB, BLOB, BFILE и ADT. Если у флага указать значение FALSE, которое стоит по умолчанию, то в случае неподдерживаемого столбца, программа логгирования завершится.

Данный пакет имеет ряд недостатков. Во-первых, он узконаправленный, сохранение информации об ошибках в не DML операциях, затруднено. Во-вторых, для логирования в нестандартную таблицу, потребуется каждый раз указывать много параметров. В-третьих, мы не можем контролировать информацию, которая будет заносится в таблицу.

В связи с вышеперечисленным, разработчикам приходится разрабатывать собственный механизм для сохранения информации о возникающих ошибках, специализированный под нужды конкретного кода.

1.7. Выводы

В ходе данной главы была рассмотрена основная информация об ошибках в PL/SQL. Были изучены инструменты для объявления пользовательских ошибок, для привязки исключительных ситуаций к кодам ошибок, были рассмотрены способы инициирования ошибок и их обработки.

При изучение существующих механизмов обработки ошибок в PL/SQL, были замечены некоторые недостатки и неудобства при работе с ними.

ГЛАВА 2. ОБЗОР И АНАЛИЗ РАЗЛИЧНЫХ СИСТЕМ ОБРАБОТКИ ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Данная глава посвящена разбору существующих проблем в системе обработок ошибок, предлагаемой кампанией Oracle.

В параграфе 2.2 рассматриваются разработки разных авторов, нацеленные на решение схожих проблем.

Про то, как происходит работа с исключительными ситуациями рассказывается в параграфе 2.3. В параграфе 2.4 приведен краткий обзор механизма взаимодействия с ошибками в объектно-ориентированных языках, на примере C#.

Описание функциональных возможностей, которые необходимы в разрабатываемом пакете, перечисляются в параграфе 2.5.

2.1. Существующие проблемы обработки ошибок

Несмотря на то, что существующая система имеет довольно много возможностей для разработчиков, она не лишена недостатков. Далее будут рассмотрены некоторые проблемы, с которыми сталкиваются программисты при разработке программного обеспечения для базы данных.

Дублирование кода. Довольно часто встречаются ситуации, в которых возможно возникновение одних и тех же ошибок, для обработки которой придется воспроизводить уже написанный код. Дублирование кода противоречит принципу разработки программного обеспечения DRY (Don’t repeat yourself, рус. не повторяйся), предложенного Энди Хайдом и Дэйвом Томасом в книге «Программист-прагматик»[4]. Увеличение кодовой базы приводит к увеличению затрат ресурсов при отладке и корректировки приложения. Одним из способов решения данной проблемы является вынесение повторяющегося кода в отдельные процедуры или функции, и заменой повторяющегося кода на вызов необходимых подпрограмм.

Очень явно данная проблема выражена при работе с пакетом UTL_FILE, а конкретнее, с процедурой GET_LINE. Данная процедура предназначена для извлечения строк из файлов, и в случае невозможности считать новую строку, будет выдано исключение NO_DATA_FOUND. Это приводит нас к тому, что каждый раз, когда нам необходимо считать данные из файла, мы должно написать код для обработки этой ошибки, который будет везде одинаковый. Данный код будет

затруднять понимание основной логики, а допущенная ошибка при копировании кода, может привести к нежелательным последствиям[9].

```
PROCEDURE read_file_and_do_stuff (
        dir_in IN VARCHAR2, file_in IN VARCHAR2
)
IS
        l_file UTL_FILE.file_type;
        l_line VARCHAR2 (32767);
BEGIN
        l_file := UTL_FILE.fopen (dir_in, file_in, 'R', max_linesize => 32767);

        LOOP
                UTL_FILE.get_line (l_file, l_line);
                ... -- Обработка считанных данных
        END LOOP;
EXCEPTION
        WHEN NO_DATA_FOUND
        THEN
                UTL_FILE.fclose (l_file);
                ... -- Дополнительная работа
END;
```

Рис.2.1. Пример использования UTL_FILE

На рис.2.1 показан пример процедуры для чтения и обработки файла, с использованием пакета UTL_FILE. Половину кода занимает обработчик ошибок, который должен присутствовать и обычно будет выполнять одну и ту же работу. Все это можно было бы сокрыть за одной функцией и работать с файлами при помощи вызова нужной подпрограммы, что займет всего одну строку кода, но позволит сократить количество ошибок и сэкономить время, затраченное на написание кода. В более сложных ситуациях, при которых использование процедуры такого вида невозможно по тем или иным причинам, все также можно будет воспользоваться пакетом UTL_FILE.

Другой немаловажной проблемой при работе с ошибками является тот факт, что EXCEPTION в PL/SQL это особая разновидность структуры данных. Мы имеем достаточно небольшой набор возможностей для работы с ней, после объявления переменной данного типа, мы можем только инициировать или обрабатывать ее. Нету возможности для расширения функциональности данной структуры, мы не можем добавить дополнительные поля, мы не можем передавать исключения как параметр для процедуры или функции.

Также Oracle не дает возможностей для организации и классификации исключительных ситуаций, относящихся к конкретному приложению. В нашем распоряжении находиться 1000 кодов в диапазоне от -20999 до -20000. Разработчику необходимо следить чтобы коды использовались правильно, разные ошибки

не должны связываться с одним и тем же кодом[3]. При боольшом количестве разработчиков это может быть затруднительно.

Для решения последних двух проблем можно реализовать альтернативную систему. Мы можем создать свои исключения и работать непосредственно с ними. Например, перечислить ошибки в отдельной таблице, тогда мы сможем передавать ошибки как параметры, дополнять всеми необходимыми для нас атрибутами, а необходимость контролировать это все перейдет от программиста к пакету, который будет этим заниматься. Под собственными ошибками будут скрыты исключения PL/SQL, в таком случае, мы, не потеряв преимуществ от работы со стандартных исключениями, будем иметь возможность при необходимости расширить их функциональность[2].

Дополнить такой пакет можно множеством различных способов, мы имеем полный контроль над аудитом ошибок, и можем его корректировать под свои нужды, в зависимости от приложения, можно собирать необходимый контекст возникновения ошибки и дополнять его при необходимости. Расширение пакета подпрограммами для обработки часто возникающих исключительных ситуаций позволит избежать проблемы дублирования кода.

2.2. Альтернативные возможности для обработки исключительных ситуаций

С обозначенными проблемами сталкивались и другие разработчики. Рассмотрим какие существуют способы улучшения процесса обработки ошибок.

Стивен Ферштейн в своем докладе [5] про хорошие практики программирования на PL/SQL упоминает про Quest Error Manager (QEM). Получить много информации о нем не удалось, так как его перестали обновлять еще в 2010 году. Удалось выяснить, что этот довольно старый пакет, разработанный кампанией Quest, предназначался для решения схожих проблем. Установлено, что целью данного пакета было упрощение логирования информации об ошибках.

В книге [3] Ферштейн и Прибыл выделяют похожие трудности при работе с исключительными ситуациями, и для изучения предоставляют пакет для обработки ошибок. Они говорят о том, что пакет не дописан и требует улучшений со стороны читателей, и предлагают выполнить это в качестве упражнения. Это небольшой пакет, состоящий всего из трех функций, предназначенный для демонстрации того, каким образом можно обрабатывать исключительные ситуации, альтернативным от стандартного способом.

В ходе поиска существующих решений, было замечено, что во многих компаниях разрабатываются собственные пакеты для упрощения работы с обработчиками ошибок, но так как эти разработки являются собственностью кампаний, в открытом доступе их нет.

2.3. Механизмы обработки ошибок в других базах данных

Рассмотрим, как исключительные ситуации обрабатываются в других базах данных.

В базе данных MySQL схема обработки ошибок довольно сильно схожа с той, которая применяется в Oracle Database. Здесь используются обработчики, которые, по своей сути, не отличаются от блока обработки ошибок в PL/SQL. Используется сигнальная схема, исключения именуются сигналами, тоже привязываются к кодам ошибок. Сигналы могут содержать дополнительную информацию об возникшей проблеме, которую можно указать при вызове сигнала, таковой информацией является: каталог, схема и имя ограничения, каталог, схема, колонка и название таблицы, имя курсора.

Обработчики бывают разных типов, например, continue и exit. Они отличаются тем, что будет происходить, после того как сигнал обработается.

В целом, такая система мало отличается от системы принятой в Oracle Database, и подвержена тем же проблемам, но имеет ряд возможностей упрощающих работу с ошибками и ускоряющих процесс написания кода[7].

Процедурный язык Transact-SQL, являющийся расширением языка SQL, разрабатывается кампанией Microsoft. Данный язык используется в базах данных SQL Server и Sybase. В Transact-SQL исключительные ситуации обрабатываются посредством блока TRY. . . CATCH. Вызов ошибок происходит посредством команды RAISERROR, в которую передаются системные параметры ошибки, такие как код, сообщение, важность и положение. Последний параметр позволяет конкретизировать место возникновение ошибки, если возникло несколько одинаковых ошибок в разных местах[8].

В базе данных PostgreSQL то, как вы будете обрабатывать ошибки, зависит от используемого языка процедурного расширения. Согласно документации, PL/pgSQL очень схож с процедурным языком Oracle PLSQl[1].

Действительно, синтаксис работы с ошибками, мало отличается. Используются такие же конструкции для блока обработки, обращаться к ошибкам можно по

имени или коду. Можно получать расширенную информацию об ошибке, схожую с информацией, передаваемой с событиями в MySQL.

В общем, обработка исключений работает схоже в разных базах данных. Везде имеется завязка на коды ошибок, корректную работу с которыми должен контролировать программист. Расширение объектов типа EXCEPTION ограничено или вовсе отсутствует.

2.4. Примеры обработки ошибок в объектно-ориентированных языках

Рассмотрим, как происходит работа с исключительными ситуациями в других языках программирования. Рассматривать будем язык C#. Изучим пример, приведенный на рисунке 2.2.

```
public class CustomException : Exception
    {
        public int CustomData;

        public CustomException(int customData)
        {
            CustomData = customData;
        }

        public void PrintData()
        {
            Console.WriteLine(CustomData);
        }
    }

    public class Program
    {
        public static void Main()
        {
            try
            {
                throw new CustomException(10);
            }
            catch (CustomException e)
            {
                e.PrintData();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.GetType());
            }
            finally
            {
                Console.WriteLine("Block finally");
            }
        }
    }
```

Рис.2.2. Пример обработки ошибок в C#

В данном коде объявляется пользовательская исключительная ситуация CustomException, она наследуется от базового класса Exception. В данном объекте

мы объявляем переменную CustomData для хранения необходимой информации, а также метод PrintData, для вывода данной информации в консоль.

Далее, в классе Program, в методе Main, происходит основная работа программы. Блок try/catch позволяет выполнить код, и в случае возникновения ошибки обработать ее. Мы имеем возможность выполнить различный код, в зависимости от ошибки, указывая несколько блоков catch (схожим образом мы разделяем обработку разных ошибок в PL/SQL). Аналогом блока try/catch в PL/SQL будет являться объявление вложенного блока begin/end с указанием ключевого слова exception.

В коде, расположенном в блоке try создается экземпляр класса пользовательской ошибки, с указанием конкретных данных, в данном случае числа 10, и происходит вызов данного исключения, при помощи ключевого слова throw. В операторе catch выполняется обработка данной ошибки, вызывается созданный нами метод, который выведет информацию на консоль.

Также, в данном примере, используется ключевое слово finally. Код, расположенный в данном блоке, выполнится независимо от того возникло какое-либо исключение или нет. Данная функциональность обычно используется для корректного завершения работы кода из блока try, например, для закрытия файлов, которые были ранее открыты.

В результате выполнения данного примера, на консоль будет выведенно две строки, первая, с числом 10, и вторая со строкой "Block finally".

Данный подход к работе с исключительными ситуациями не накладывает на нас ограничений, которые существуют в PL/SQL. Программисту не нужно следить за кодами ошибок, имеется достаточное количество предопределенных классов, с различными ошибками, мы можем спокойно передавать любое количество данных. Современные IDE подскажут разработчику какие исключения созданы, и уже из представленного списка, можно будет выбрать подходящее.

В других объектно-ориентированных языках взаимодействие с исключительными ситуациями происходит схожим образом.

2.5. Требования к разрабатываемому пакету

Среди выдвинутых проблем при обработке ошибок, особенно выделяется неудобство взаимодействия с кодами ошибок. Разработчики продукта обязаны самостоятельно контролировать и следить за тем, какие коды использованы, какие

свободны, какой номер ошибки нужно применить в той или иной ситуации. Не редко для этих целей используется документ, в котором программисты составляют перечень номеров, которые уже используются. Для решения этой проблемы в пакете будет присутствовать таблица, содержащая подробную информацию об каждой ошибке. В которую входит: код ошибки, тип ошибки, имя для исключения, подробное описание исключительной ситуации, информация об области применения, дополнительная информация, специфичная для пользователей пакета.

Особенно важной функциональностью для такого пакета является функциональность сохранения информации об возникающих ошибках. А также различные представления, предоставляющие удобный способ для анализа ошибок. Должна быть собрана информация об частотности возникновения ошибок, местах их появления, данных, которые привели к возникновению ошибке такого рода.

Для избавления проблем с дублированием кода, будут предусмотрены несколько стандартных методов-обработчиков ошибок, с возможностью расширить тот список пользовательскими обработчиками. Среди стандартных обработчиков должны присутствовать:

A. Обработчик с логированием и повторным возбуждением исключения, заносящий информацию в специальное место, и передающий ошибку в вызывающий блок.
B. Обработчик с логированием и сокрытием ошибки, данный метод будет сохранять информацию и завершать выполнение блока, в котором возникло исключение, без передачи ошибки дальше.
C. «Тихие» обработчики, данные обработчики схожи по функционалу с предыдущими, но не производят логирования об возникшей ошибке.
D. Обработчик критических ошибок, такой метод предназначен для работы с особо важными ошибками, затрагивающими основную логику приложения. Помимо занесения информации в стандартные места, данная процедура позволит сообщить администратору об возникновении критической ситуации, через другие каналы связи, например по почте. Функционал для настройки оповещения будет предусмотрен в разрабатываемом пакете.

Необходима возможность для настройки параметров пакета, так называемые feature flag. Это поля (обычно содержащие бинарное значение), позволяющие включить или отключить, ту или иную функциональность.

Данный список требований будет пополнен функциональными возможностями, необходимость в которых возникнет в ходе разработки.

2.6. Выводы

В ходе данной главы были рассмотрены проблемы, которые затрудняют разработку продуктов на языке Oracle PL/SQL. Были предложены различные способы для их устранения. Был проведен разбор пакетов, упрощающих работу с исключительными ситуациями. Также было рассмотрено как обработка ошибок реализована в других СУБД. После этого были выдвинуты требования к реализуемому пакету, с описанием того, как та или иная функциональность способствует решению обозначенных проблем.

ГЛАВА 3. РАЗРАБОТКА ПАКЕТА

В данной главе рассматривается различные этапы создания пакета. Начиная с разработки и выборов алгоритмов работы с ошибками, заканчивая созданием скриптов для создания и удаления пакета.

3.1. Настройка среды

Для разработки будет использоваться база данных Oracle версии 11g Release 2. Установленная в виртуальную машину, Oracle VirtualBox, под управлением ОС Windows Server 2008. Дополнительно нужно установить схемы примеров, предоставляемые Oracle (в дальнейшем потребуется для тестирования), а также пакет UTL_MAIL для отправки Email сообщений, который поставляется с базой, но не установлен по умолчанию. В качестве среды разработки будем использовать Oracle SqlDeveloper и JetBrains DataGrip. Создадим пользователя, от лица которого будем разрабатывать пакет.

```
create user diploma
identified by diploma
profile default
default tablespace diploma
quota unlimited on diploma
account unlock;

grant connect to diploma;
grant create table to diploma;
grant create view to diploma;
grant create procedure to diploma;
grant create trigger to diploma;
grant create sequence to diploma;
grant create synonym to diploma;
grant create public synonym to diploma;

grant create any directory to diploma;
grant drop any directory to diploma;

grant execute on UTL_FILE to diploma;
grant execute on UTL_MAIL to diploma;
```

Рис.3.1. Код создания пользователя и выдачи привилегий

На рисунке рис.3.1 представлен скрипт создания пользователя и выдачи привилегий. Пользователю потребуются: привилегий на создание подключения, создание таблиц, представлений, синонимов и публичных синонимов, последовательностей, процедур и триггеров, привилегии для создания и удаления директорий, а также привилегии на работу с пакетами UTL_FILE и UTL_MAIL.

Данный пользователь предназначен только для разработки, пользователям пакета не нужно будет создавать такого пользователя, поэтому использование некоторых потенциально опасных привилегий (например, drop any directory) является допустимым. Если в ходе разработки потребуются дополнительные разрешения, об этом будет специально указано.

3.2. Схема установки пакета

Пакет будет распространяться как SQL скрипт, который состоит из установщика и скриптов самого пакета. Инсталлер выполняет работу по созданию основных объектов, таблиц, триггеров, последовательностей, представлений. После чего производится вставка данных, в главные таблицы, сюда входит информация об основных ошибках, описание их типов, создание параметров и выставление их в значения по умолчанию. Так же установочный скрипт занимается созданием директории для логирования файлов.

3.3. Схема хранения информации

Первоначально, для решения проблемы работы с кодами ошибок, необходимо сокрыть их использование за процедурами пакета. Опишем сущность исключительной ситуации, с которой будем работать в дальнейшем. Для ошибки нам необходимо хранить следующую информацию: код ошибки, наименование ошибки, тип ошибки и информацию о ней, а также дополнительную мета информацию, в которую будет входить, временная метка создания записи об ошибки, и имя пользователя, который создал данную ошибку. Номер ошибки (код) будет использоваться в коде пакета, для упрощения идентификации, пользователи же будут использовать наименования ошибки, так как правильно выбранное имя, более подробно описывает суть ошибки и проще для запоминания человеком, но, в случае необходимости, пользователи все также могут обращаться к ошибкам по кодам. Это позволит сделать разрабатываемый пакет расширяемым, не скрывая от пользователей всех возможностей. Данная информация будет храниться в таблице ERRM$ERRORS, которая будет связана с таблицей ERRM$ERROR_TYPES. Таблица ERRM$ERROR_TYPES будет содержать описание типов исключительных ситуаций. Типы позволят разделить ошибки на группы, и обрабатывать ошибки по-разному в зависимости от их важности. Информация, хранящаяся о категориях,

подобна той информации, которую мы храним об ошибках, здесь содержится ID типа, наименование типа и информация о нем.

Модель хранения данных, используемая в пакете, представлена на рисунке рис.3.2.

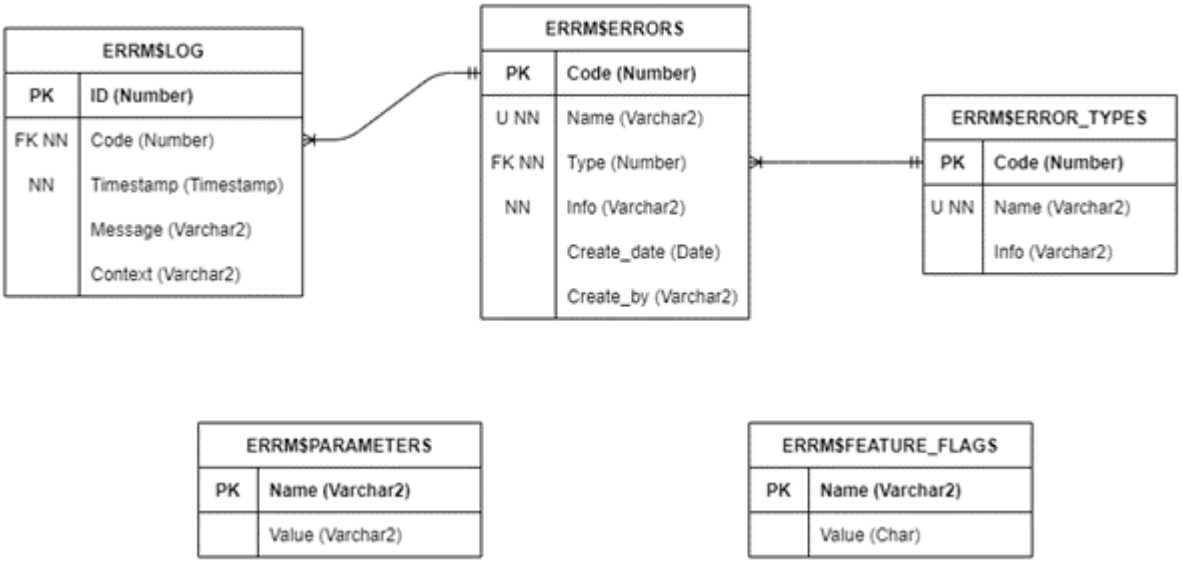

Рис.3.2. ER модель хранения информации пакета

Здесь и далее, для Entity-relationship моделей используются следующие обозначения: PK – первичный ключ (Primary Key), FK – внешний ключ (Foreign Key), U – уникальное поле (Unique), NN – обязательное поле (Not null).

Таблица ERRM$ERRORS имеет ссылочное ограничение на первичный ключ таблицы ERRM$ERROR_TYPES. Возникающий в ходе работы приложения, ошибки необходимо хранить для дальнейшего анализа, для данных целей присутствует таблица ERRM$LOG. В которую заносится номер записи, код ошибки (имеется связь с таблицей ERRM$ERRORS), временная метка возникновения ошибки, сообщение и контекст данной ситуации. Здесь, и в дальнейшем, под контекстом ошибки понимается набор наименований каких-либо данных и связанных с ними значений, если не сказано иного.

Дополнительно, присутствуют две таблицы для задания параметров, это ERRM$PARAMETERS и ERRM$FEATURE_FLAGS. Вторая таблица хранит параметры для отключаемых настроек, то есть тех настроек, которые имеют бинарное значение (включена/выключена), например возможность логирования информации в файл. А первая отвечает за настройку не бинарных параметров, например, название файла для записи информации об ошибках, почта администратора.

3.4. Дополнительные объекты для обработки информации

Помимо перечисленных в параграфе 3.3 таблиц, нам необходимы дополнительные объекты базы данных, которые будут обеспечивать целостность и корректность хранимой информации.

В первую категорию обозначенных объектов входят последовательности. Последовательности – это объект базы данных, предоставляющий пользователю последовательность чисел по указанным правилам. Нами будет использоваться три последовательности. Первая – для кода ошибок, вторая – для номера типов ошибок, и третья – для последовательного номера записи в таблицу логирования.

Во вторую категорию попадают триггеры на таблицы. Для таблиц ERRM$PARAMETERS и ERRM$FEATURE_FLAGS за счет триггеров будет контролироваться вид хранимой информации, все имена параметров должны храниться в верхнем регистре, а значения параметров в нижнем, значения для feature flag должны быть либо символом 0, либо символом 1. В случае, если пользователи будут добавлять некорректные данные, триггер не позволит вставить такую информацию, обновление информации будет отклонено, возникнет исключение с сообщением.

3.5. Определение спецификации пакета

Весь пакет (спецификация и тело) будет образно разделен на подкатегории, таким образом пользователям будет проще ориентироваться в пакете, читаемость кода такого пакета будет выше. Были выделены следующие категории структуры пакета:

A. CONSTANTS – в данном блоке будут объявляться константные значения пакета
B. TYPES – объявление используемых типов данных
C. PARAMETERS WORK – процедуры и функции для настройки параметров пакета
D. HELPERS – дополнительные действия
E. PROCEDURE AND FUNCTIONS – методы для основной работы с пакетом
F. SILENT RAISES – методы для вызова исключений без логирования
G. LOG WORK – процедуры работы с местами хранения логов

H. SIMPLE ERROR RAISE – процедуры для упрощения вызова часто используемых ошибок
I. LOG – блок для вывода информации на экран, упрощение работы с пакетом DBMS_OUTPUT

В коде пакета данные категории будут помечены комментариями и разделяться пустыми строками. Данное разделение является условным и на работу программы не влияет.

Выделим типы, с которыми будем работать. Объявив типы (или подтипы, как в нашем случае) мы сможем отвязаться от конкретного способа хранения данных, и, в случае кода тип нужно будет поменять, например если код ошибки потребуется хранить не числом, а строкой, это сделать будет очень просто, поменяв объявление типа в спецификации пакета. Мною были объявлены следующие типы данных:

A. t_err_code – тип для идентификации ошибки, является подтипом типа pls_integer
B. t_err_name – тип для работы с именем ошибки, подтип varchar2 размером в 100 символов.
C. t_err_type – тип для хранения категории ошибки, наследуется от типа pls_integer

Для возможности настройки параметров пакета нам потребуется, как минимум, 4 метода. Получение и задания значений параметров, а также получений и задание флагов возможностей. Оба метода на получение установленного значения будут принимать один входной параметр – имя настройки, а возвращать сохраненное в таблице значение, отличие будет в возвращаемом типе данных, для параметров это будет строка (varchar2), для опций это Boolean.

Так же для работы с опциями и параметрами объявим несколько строковых констант, в которых будут сохранены имена для параметров. Это позволит пользователям не заботиться о правильности написания названия того или иного параметра, а просто обратиться к константе.

Для констант будут использоваться следующие префиксы: FEATURE_ для бинарных опция (которые хранятся в таблице ERRM$FEATURE_FLAGS, подробнее смотри в параграфе 3.3) и PARAMETER_, собственно, для параметров (хранятся в таблице ERRM$PARAMETERS). При помощи этих префиксов будет понятно, какие методы нужно использовать, и какое значение мы ожидаем в результате работы функции.

```
declare
    l_log_file BOOLEAN;
begin
    l_log_file := errm.GET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_FILE);
    errm.LOG(case when l_log_file then 'TRUE' else 'FALSE' end);
    errm.SET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_FILE, false);
end;
/
```

Рис.3.3. Пример получения установленного значения опции

На рисунке рис.3.3 показан пример использования описанных ранее процедур и функция для работы с параметрами. В данном анонимном блоке объявляется бинарная переменная, в которую заносится значение опции FEATURE_LOG_ERRORS_TO_FILE, которая отвечает за логирование в файл. После чего происходит вывод информации на экран, при помощи метода LOG (о нем будет сказано далее), затем происходит отключение возможности логирования в файл.

Использование констант также защищает пользователей от дополнительных ошибок, если бы пользователю пришлось задавать название параметра вручную, велика вероятность опечатки, и в результате мы получим ошибку на этапе исполнения, когда программа сделает запрос к таблице и не найдет параметра с таким значением. В случае же использования констант среда разработки подскажет, какие константы мы можем использовать, но даже в случае опечатки, ошибка вылезет еще на этапе компиляции, что позволит заметить ее гораздо раньше, и исправить. При этом функции GET_FEATURE_VALUE и GET_PARAMETER_VALUE принимают строки, что дает возможность пользователям, в случае крайней необходимости, обратиться к параметру не через константу.

Методы для вывода информации из категории LOG, сюда входит две процедуры LOG (которая уже использовалась ранее) и NL. Данные методы являются обертками для вызовов стандартных методов из пакета DBMS_OUTPUT. Процедура LOG работает аналогично, процедуре PUT_LINE, а процедура NL скрывает за собой процедуру NEW_LINE. Это упрощает написание кода для отладки, и позволяет с легкостью заменить вывод отладки, например, на вывод в файл, достаточно будет поправить только один метод. Нет нужды заставлять пользователей использовать данные методы, так как не известно как конкретно работает логирование в пользовательском приложении, довольно вероятна ситуация, когда используется другой пакет, например, log4plsql, но для удобства они были вынесены в спецификацию.

В категорию HELPERS входит единственный метод GET_ERROR_CODE, который позволяет по имени ошибки получить ее код. Методы для пользователей имеют определения как для работы с кодами ошибок, так и для работы с именами ошибок (что является более предпочтительным), но данная функциональность может потребоваться пользователям пакета. Допустим, что имеется необходимость не вызывать ошибку прямо сейчас, а требуется записать ошибку, либо передать ее куда-то, выполнить дополнительную работу, и только потом вызвать исключение, в таком случае можно сохранить (или передать в другой метод) имя ошибки, в котором произойдет необходимая работа, после чего, при помощи данного метода происходит получение кода ошибки, и проводятся дополнительные операции с ошибкой, и в итоге происходит вызов ошибки. Кратко говоря, данная функциональность расширяет перечень возможностей пакета.

Рассмотрим процедуры для работы с местами логирования информации (категория LOG WORK). Сюда включены две процедуры CLEAR_LOG_TABLE и CLEAR_LOG_FILE. Как понятно из их названия, они предназначены для очистки хранилища ошибок, таблицы или файла, соответственно.

Далее будут рассмотрены методы для основной работы с пакетом.

Процедура REGISTER_ERROR предназначена для создания ошибки, за ней будет скрываться вставка данных в таблицу ERRM$ERRORS, предваренная дополнительными действиями, генерация кода ошибки, если он не указан, приведение имени ошибки к стандартному имени и так далее. На вход процедура принимает: код ошибки, не обязательный параметр, наименование ошибки (обязательно), тип ошибки (не обязательно, по умолчанию устанавливается обычная ошибка), информация об ошибке (обязательно), комментарий про использование данного вида ошибки. Информация об ошибке фактически не является обязательным параметром, но будем заставлять пользователей ее указывать, чтобы при возникновении ошибки была максимально понятна ее суть. Поля с датой создания и именем пользователя-создателем, будут обновляться при помощи триггера. Для вызова ошибки будут использоваться методы RAISE, всего таких методов будет 4. Первый (самый правильный вариант) принимает в параметрах имя вызываемой ошибки, и не обязательный флаг p_log_enabled, который отвечает за необходимость логирования информации об ошибке. Второй метод, принимает вместо имени код ошибки, имеет аналогичный второй параметр. В обоих этих методах параметр p_log_enabled не является обязательным и по умолчанию имеет значение true, что означает, что логирование будет производиться.

Остальные два метода имеют префикс s (сокращение от слова silent(англ.) – тихий), итоговое название SRAISE, они вызывают обычный метод raise, с установкой второго параметра в значение false. Что означает, что логирование производиться не будет. Данный обработчик используется в тех случаях, когда нужно прервать выполнение программы, но не требуется заносить информацию об этом. Методы по умолчанию принимает имя ошибки, но имеет перегрузку процедуры, в которой задается номер ошибки, вместо имени. В категорию SIMPLE ERROR RAISE попадают процедуры, которые позволят вызвать исключение одной строчкой, не нужно знать даже наименование ошибки, что очень удобно для часто используемых исключительных ситуаций. В данный момент, сюда включено 4 метода, для заранее предопределенных ошибок: DEFAULT_ERROR, DEFAULT_LOG, DEFAULT_WARNING и NOT_IMPLEMENTED_EXCEPTION. Первые три представляют собой вызов стандартных (для данного пакета) исключительных ситуация с разными типами (подробнее смотри в параграфе 3.3), последний является вызовом ошибки, которая гласит, что код еще не реализован.

Рассмотрим пример, пользователь разрабатывает пакет, определил спецификацию и реализовал в теле не все методы, но имеется желание уже протестировать работу этих пакетов. Для нереализованных методов можно создать простую реализацию с заглушкой состоящей из одной команды null; , но такая ситуация является опасной, если метод так и не будет реализован, то при вызове этих методов ничего не произойдет, опасным является то, что об этом никто не узнает, кто-то ожидает, что метод выполнит свою работу, но этого не происходит. Если же использовать исключение, то при вызове данного метода, сразу станет понятно, что метод еще не готов, и использовать его еще рано. Так же это упростит автоматическое тестирование.

Рассмотрим, как будет проходить взаимодействие пользователей с контекстом ошибок. Для этого в пакете предусмотрены две процедуры ADD_CONTEXT и CLEAR_CONTEXT. Первая добавит к контексту дополнительную информацию, принимает два строковых параметра: название и значение. Вторая очищает значения в сохраненном контексте. Контекст автоматически очистится после вызова ошибки.

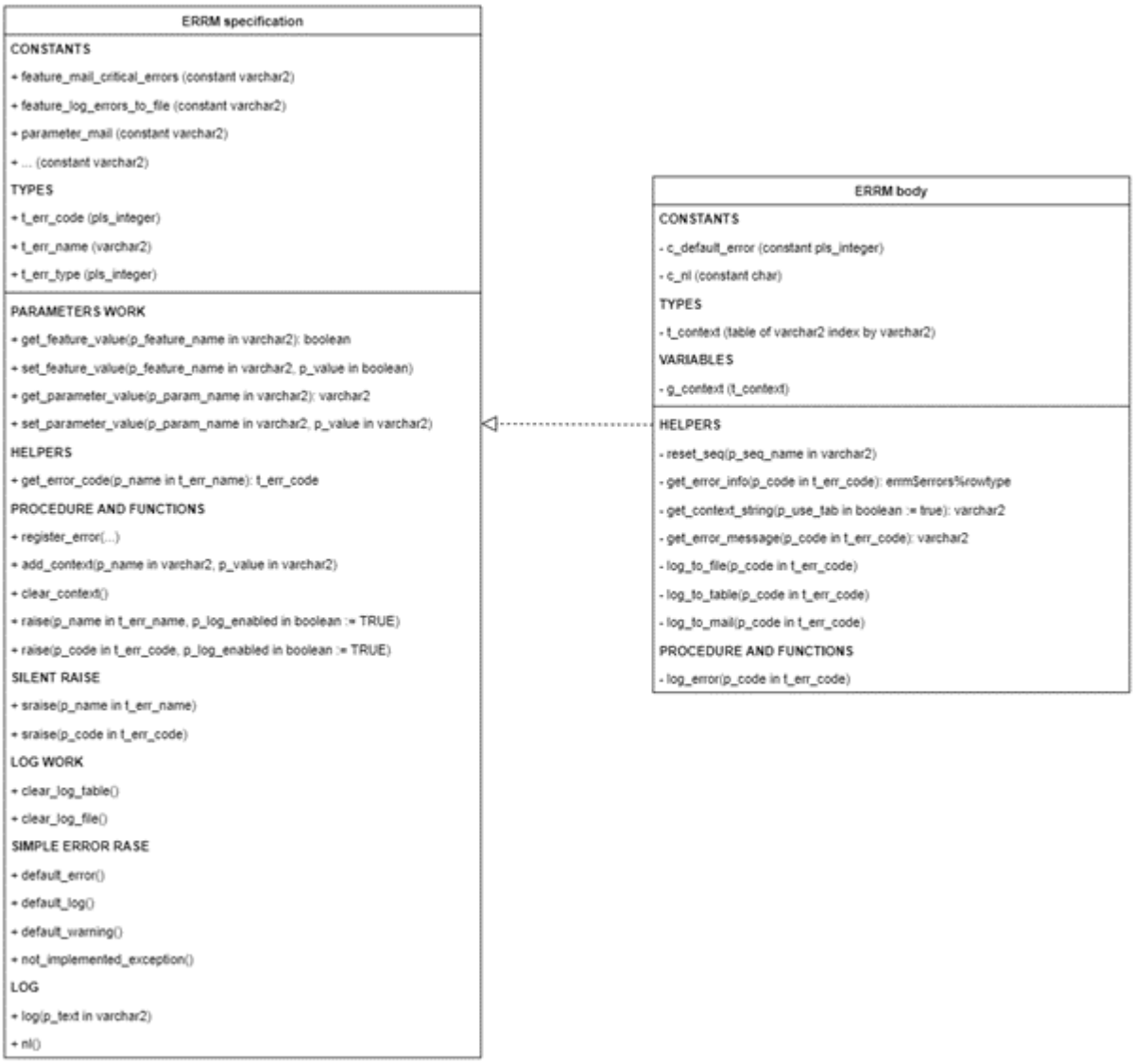

Рис.3.4. UML диаграмма пакета

На рисунке рис.3.4 представлена схематичная UML диаграмма пакета. В теле пакета указаны только методы, отличные от спецификации пакета, остальные методы, тоже будут реализованы. Категории блоков пакета выделены жирным.

3.6. Создание установщика

Весь пакет состоит из четырех файлов: установщик, определение спецификации пакета, определение тела пакета, деинсталлер. Установщик занимается созданием и настройкой необходимых объектов, после чего запускает скрипты для создания пакета. В итоге три первых скрипта могут быть объединены в один, для исключения лишних ошибок.

Все начинается с создания таблицы типов ошибок ERRM$ERROR_TYPES. Код этой части инсталлера представлен на рисунке рис.3.5.

```
--CREATE ERROR_TYPES TABLE
create table errm$error_types (
    code number(5) primary key,
    name varchar2(100 char) unique not null,
    info varchar2(2000 char)
);

create sequence errm$errors_type_srq start with 4 increment by 1;

insert into errm$error_types values (0, 'error', 'Exception');
insert into errm$error_types values (1, 'warning', 'Warning');
insert into errm$error_types values (2, 'log', 'Log information');
insert into errm$error_types values (3, 'critical_error', 'Exceptions that
require additional actions');
```

Рис.3.5. Код создания таблицы типов

Первоначально создается таблица для хранения типа ошибок, затем создается последовательность для хранения кодов типов, после чего производится вставка нескольких базовых типов.

```
--CREATE ERRORS TABLE
create table errm$errors (
        code number(10) primary key,
        name varchar2(100) unique not null,
        type number(5) default 0 references errm$error_types not null,
        info varchar2(2000) not null,
        usage_comment varchar2(1000),
        create_date date,
        create_by varchar2(100)
);
/

create sequence errm$errors_code_seq start with 4 increment by 1;

create or replace trigger errm$additional_info_biu_trg
    before insert or update
    on errm$errors
    for each row
begin
    :new.name := lower(:new.name);
    if :old.create_date is null then
        :new.create_date := sysdate;
        :new.create_by := SYS_CONTEXT ('USERENV', 'SESSION_USER');
    end if;
end errm$additional_info_biu_trg;

insert into errm$errors values (0, 'default_error', 0, 'Error has occurred',
'Simple error for tests. Do not use it in production!', null, null);
insert into errm$errors values (1, 'not_implemented', 0, 'Code not
implemented', 'Use when code not yet implemented.', null, null);
insert into errm$errors values (2, 'default_warning', 1, 'Warning
information', 'Simple warning for tests. Do not use it in production!', null,
null);
insert into errm$errors values (3, 'default_log', 2, 'Log information',
'Simple log message for tests. Do not use it in production!', null, null);
```

Рис.3.6. Код создания таблицы ошибок

На рисунке рис.3.6 представлен код создания таблицы ошибок, создания последовательности. Создается триггер, который приводит имя ошибки к стандартному виду и добавляет мета информацию и создании. Для этого используется стандартная процедура SYS_CONTEXT, которая позволяет получить имя пользователя.

```
--CREATE LOG TABLE
create table errm$log (
    id number(38) primary key,
    code number(10) references errm$errors not null,
    timestamp timestamp not null,
    message varchar2(1000),
    context varchar2(2000)
);

create sequence errm$log_seq start with 1 increment by 1;
```

Рис.3.7. Код создания таблицы для логирования

На рисунке рис.3.7 показан код создания таблицы для хранения логов ошибки, и последовательности, для нумерации логов.

Код для создания остальных таблиц ERRM$PREFERENCES и ERRM$FEATURE_FLAGS, а также триггеров и данных, можно посмотреть в приложении 1.

3.7. Разработка тела пакета

Далее будут подробно описаны принципиально важные участки пакета, менее важные моменты рассмотрены не будут, полный код пакета содержится в приложении 1. Схематично вызов исключения выглядит как на рисунке рис.3.8.

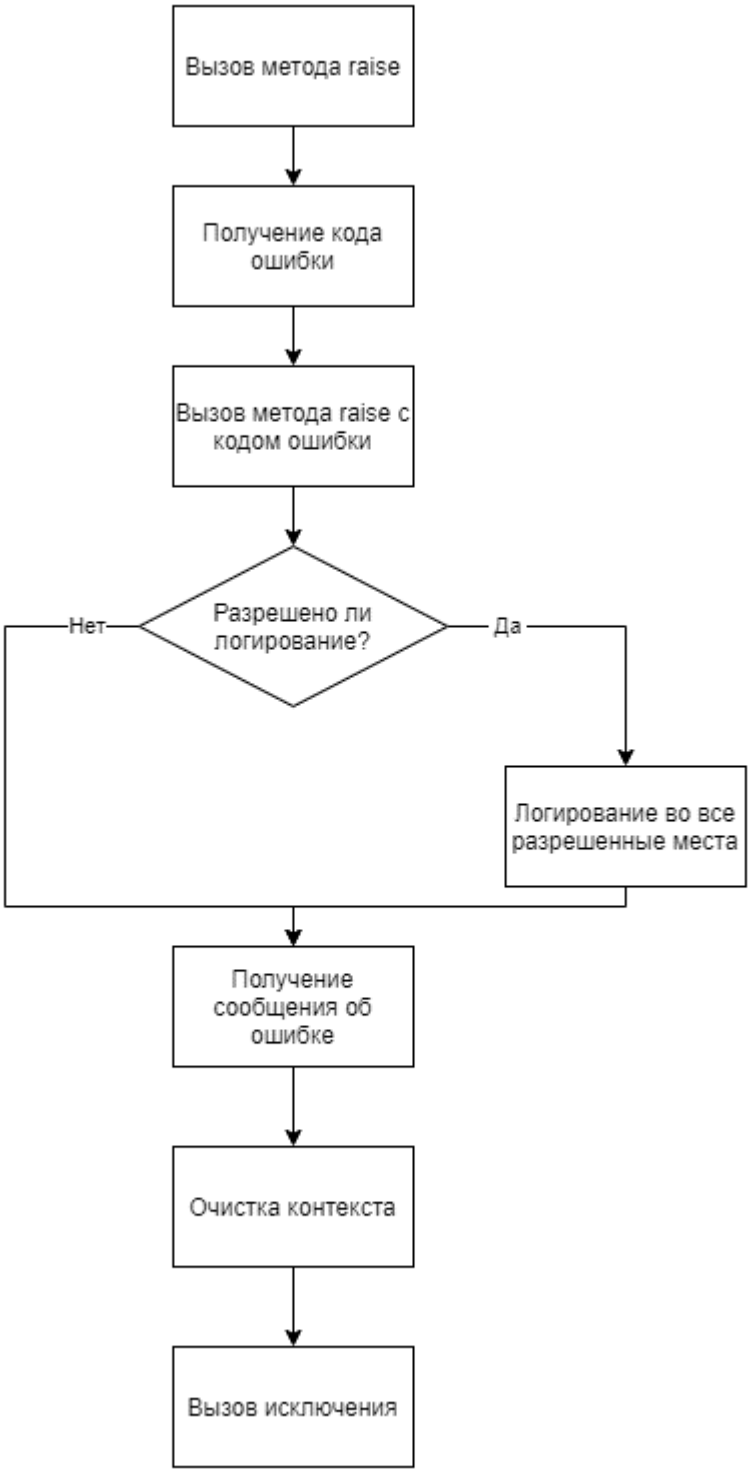

Рис.3.8. Блок схема вызова исключения

Первую операцию выполняет пользователь, остальное скрыто за реализацией пакета. В случае логирования информации об ошибке, каждая процедура для логирования информации в определенное место (например, на почту), проверяет доступность этого метода согласно установленным параметрам пакета.

Вызов исключения всегда происходит с определенным кодом, который задается при создании тела пакета, используется значение -20000, зарезервированное Oracle для пользовательских приложений. Вызов исключения происходит при помощи стандартной процедуры RAISE_APPLICATION_ERROR, в который передается указанный код, а также отформатированное сообщение об ошибке, в нем указывается название и код исключительной ситуации из пакета, краткое описание ошибки, тип ошибки и контекст. В случае, если включено все логирование эта же информация заносится в таблицу ERRM$LOG, и в файл логирования. Если возникшая ошибка является критической, происходит отправка письма администратору, почта которого задается в параметрах пакета.

Данное описание справедливо для процедуры RAISE принимающей код ошибки.

Все остальные методы для возбуждения исключения (RAISE с именем ошибки, два вида SRAISE, упрощенные вызовы частых ошибок) сводятся к вызову основного метода RAISE.

```
procedure raise(p_code in t_err_code, p_log_enabled in boolean := TRUE)
is
    l_message varchar2(2000);
begin
    if p_log_enabled then
        log_error(p_code);
    end if;

    l_message := get_error_message(p_code);

    clear_context();
    raise_application_error(c_default_error, l_message);
end raise;

procedure raise(p_name in t_err_name, p_log_enabled in boolean := TRUE)
is
    l_err_code t_err_code;
begin
    l_err_code := get_error_code(p_name);
    errm.raise(l_err_code, p_log_enabled);
end raise;

procedure sraise(p_code in t_err_code)
is
begin
    errm.raise(p_code, FALSE);
end sraise;

procedure sraise(p_name in t_err_name)
is
begin
    errm.raise(p_name, FALSE);
end sraise;
```

Рис.3.9. Код метода Raise

Код процедуры RAISE и того, как остальные процедуры завязаны на основной метод представлен на рисунке рис.3.9.

Методы для упрощенного вызова ошибок работают следующим образом: вызывается метод RAISE с указанным в методе кодом ошибки. Для расширения перечня таких методов пользователи могут добавить аналогичные методы, либо в данный пакет и перекомпилировать его, либо в какое-либо свое пространство. Данное действие рекомендуется провести для всех часто используемых ошибок.

Для логирования информации об ошибке основной метод RAISE вызывает процедуру LOG_ERROR. Которая, в свою очередь, вызывает три процедуры, каждая из который заносит информацию об ошибке, в свое место.

```
procedure log_to_file(p_code in t_err_code)
is
    l_log UTL_FILE.file_type;
begin
    if NOT get_feature_value(FEATURE_LOG_ERRORS_TO_FILE) then
        return;
    end if;

    l_log := UTL_FILE.FOPEN('ERRM$LOG_DIR', 'log.txt','A'); -- A - append
    UTL_FILE.PUT(l_log, systimestamp || ': ' || get_error_message(p_code));
    UTL_FILE.FCLOSE(l_log);
end log_to_file;

procedure log_to_table(p_code in t_err_code)
is
    pragma autonomous_transaction;

    l_error errm$errors%rowtype;
    l_message errm$log.message%type;
    l_context errm$log.context%type;
begin
    if NOT get_feature_value(FEATURE_LOG_ERRORS_TO_TABLE) then
        return;
    end if;

    l_error := get_error_info(p_code);
    l_message := get_error_message(p_code);
    l_context := get_context_string(false);

    insert into ERRM$LOG values (errm$log_seq.nextval, p_code, systimestamp,
l_message, l_context);
    commit;
end log_to_table;
```

Рис.3.10. Коды процедур логирования в файл и в таблицу

На рисунке рис.3.10 представлены процедуры для занесения информации в файл и в таблицу, соответственно. Каждая процедура перед логированием информации, проверяет включена ли соответствующая опция. Метод LOG_TO_FILE открывает файл при помощи функции UTL_FILE.FOPEN, с параметром A, который позволяет добавить информацию в конец файла и заносит необходимую информацию.

Метод LOG_TO_TABLE получает необходимые значения, сюда входит: номер лога, сообщение об ошибке, контекст ошибки, а затем вносит данные в

таблицу. Метод объявлен как автономная транзакция, это необходимо для того, чтобы данные в таблице ERRM$LOG не откатились вместе с основной транзакцией при возникновении ошибки (которая будет далее вызвана методом RAISE).

```
procedure log_to_mail(p_code in t_err_code)
is
    l_host varchar2(100);
    l_db_name varchar2(100);
    l_error_instance errm$errors%rowtype;
    l_message varchar2(2000);
begin
    if NOT get_feature_value(FEATURE_MAIL_CRITICAL_ERRORS) then
        return;
    end if;
    l_error_instance := get_error_info(p_code);

    if l_error_instance.TYPE != 4 then
        return;
    end if;

    l_host := SYS_CONTEXT ('USERENV', 'HOST');
    l_db_name := SYS_CONTEXT ('USERENV', 'DB_NAME');
    l_message := get_error_message(p_code);

    UTL_MAIL.send(
            sender     => l_db_name || '@' || l_host || '.com',
            recipients => get_parameter_value(errm.PARAMETER_MAIL),
            subject    => l_error_instance.NAME || ' has occurred on ' ||
l_db_name || '/' || l_host,
            message    => l_message);
end log_to_mail;

procedure log_error(p_code in t_err_code)
is
begin
    log_to_table(p_code);
    log_to_file(p_code);
    log_to_mail(p_code);
end log_error;
```

Рис.3.11. Код логирования ошибок

Код, представленный на рисунке рис.3.11 отвечает за логирование критических ошибок на почту, и общий метод логирования, который и вызывает процедура RAISE. Процедура LOG_TO_MAIL, отправляет письмо администратору, адрес которого указывается в параметрах приложения.

Процедура LOG_ERROR по очереди вызывает ранее описанные методы.

Очистка мест сбора информации происходит следующим образом. В случае хранения информации в файле (метод CLEAR_LOG_FILE), файл открывается для записи, при помощи пакета UTL_FILES, метода FOPEN, с указанием параметра W, что очистит файл для записи новой информации. После чего файл закрывается. В результате данного действия файл станет пустым.

Сложнее происходит очистка таблицы ERRM$LOG, помимо удаления данных из самой таблицы, необходимо обнулить последовательность номеров лога.

Пересоздать последовательность заново мы не можем, так как, в этом случае пакет (имеющий зависимость на эту последовательность) потребует перекомпиляции. Обнулять последовательность будем способом, показанным на рисунке рис.3.12.

```
procedure reset_seq(p_seq_name in varchar2)
is
    pragma autonomous_transaction;
    l_val number;
begin
    execute immediate 'select ' || p_seq_name || '.nextval from dual' INTO
l_val;
    execute immediate 'alter sequence ' || p_seq_name || ' increment by -' ||
l_val || ' minvalue 0';
    execute immediate 'select ' || p_seq_name || '.nextval from dual' INTO
l_val;
    execute immediate 'alter sequence ' || p_seq_name || ' increment by 1
minvalue 0';
end reset_seq;

procedure clear_log_table
is
    pragma autonomous_transaction;
begin
    execute immediate 'truncate table errm$log';

    --We can't drop and recreate our sequence, because package should be
compiled again
    reset_seq('ERRM$LOG_SEQ');
end clear_log_table;

procedure clear_log_file
is
    l_log UTL_FILE.file_type;
begin
    l_log := UTL_FILE.FOPEN('ERRM$LOG_DIR', 'log.txt','W');
    UTL_FILE.FCLOSE(l_log);
end clear_log_file;
```

Рис.3.12. Код процедур для очистки мест логирования

Здесь представлен фрагмент кода с очисткой файла логирования, и метод для очистки таблицы логирования. Сама очистка таблицы происходит посредством вызова динамического SQL, а конкретно команды TRUNCATE TABLE, после чего происходит вызов процедуры RESET_SEQ, в который передается имя последовательности. В методе RESET_SEQ происходит динамический вызов четырех команд SQL. Первоначально из последовательности извлекается следующее число, далее происходит изменение последовательности при помощи команды ALTER SEQUENCE, изменяется шаг на отрицательное значение, полученного на прошлом этапе числа. Далее извлекается следующее значение последовательности, в ре-

зультате чего текущее число последовательности обнулится. Последняя команда возвращает шаг последовательности к начальному значению.

3.8. Работа с контекстом

Для сохранения контекста ошибки будет использоваться схема, описанная далее. В теле пакета объявляется тип данных t_context, являющийся таблицей, хранящей тип varchar2 и индексируемой типом varchar2 (аналог ассоциативного массива в PL/SQL, еще называемый словарем). А также объект данного типа с именем g_context. При помощи метода ADD_CONTEXT, в данный словарь будут вноситься данные. При возникновении ошибки информация из контекста попадает в лог, ассоциативный массив очищается, для хранения контекста новых ошибок. Процедура CLEAR_CONTEXT позволяет очищать контекст вручную, в случае необходимости. Дополнительная процедура ADD_DEFAULT_CONTEXT добавляет к контексту системную информацию, в нее включены: наименование экземпляра, обслуживающего базу, наименование базы данных, имя пользователя, имя приложения из которого выполнялась работа, текущая схема, наименование хоста. Описанный базовый контекст может добавляться автоматически, ко всем возникшим ошибкам, это поведение настраивается флагом в таблице ERRM$FEATURE_FLAGS, по умолчанию включено.

```
procedure add_default_context
is
begin
    add_context('instance', SYS_CONTEXT ('USERENV', 'INSTANCE_NAME'));
    add_context('db_name', SYS_CONTEXT ('USERENV', 'DB_NAME'));
    add_context('user_name', SYS_CONTEXT ('USERENV', 'SESSION_USER'));
    add_context('module', SYS_CONTEXT ('USERENV', 'MODULE'));
    add_context('current_schema', SYS_CONTEXT ('USERENV', 'CURRENT_SCHEMA'));
    add_context('host', SYS_CONTEXT ('USERENV', 'HOST'));
end add_default_context;

procedure add_context(p_name in varchar2, p_value in varchar2)
is
begin
    g_context(p_name) := p_value;
end add_context;

procedure clear_context
is
begin
    g_context.DELETE();
end clear_context;
```

Рис.3.13. Методы добавления информации в контекст ошибки

На фрагменте кода, представленном на рисунке рис.3.13, показаны методы для добавления контекста, для очистки словаря контекста, и для заполнения словаря контекстом по умолчанию.

Так же для работы с контекстом, используется метод GET_CONTEXT_STRING, который формирует из словаря строку с контекстом, которая уже и добавляется в логи. Код данной функции представлен на рисунке рис.3.14.

```
function get_context_string(p_use_tab boolean := true)
    return varchar2
    is
        l_index   varchar2(100);
        l_context varchar2(1000);
    begin
        if get_feature_value(errm.FEATURE_ADD_DEFAULT_CONTEXT) then
            add_default_context();
        end if;

        if g_context.COUNT = 0 then
            return null;
        end if;

        l_index := g_context.FIRST;

        while l_index is not null
        loop
            l_context := l_context || case when p_use_tab then '    ' else ''
end || l_index || ': ' || g_context(l_index) || c_nl;
            l_index := g_context.next(l_index);
        end loop;

        return l_context;
    end get_context_string;
```

Рис.3.14. Функция получения строки контекста

Добавление контекста по умолчанию, происходит именно в этом методе, так как строка с контекстом обязательно будет запрошено при вызове исключения. В качестве параметра, данная функция принимает флаг, указывающий нужно добавлять дополнительный отступ перед параметрами контекста, для логирования в таблицу данный отступ не является необходимым, потому что контекст будет храниться в отдельном столбце, а для вывода на экран и в файл, отступ будет упрощать чтение сообщения об ошибке.

Данный подход для работы с контекстом почти не ограничивает пользователя в хранимом количестве данных, ограничение появляется только при достижении лимита размера типа varchar2. Максимальный размер данного типа 4000 байт, что эквивалентно 4000 символов в одно байтовой кодировке или 2000 в двухбайтовой, соответственно. Данного объема достаточно для ошибок, контекст ошибки не должен содержать больших данных и должен давать максимально подробное описание ситуации, при минимальном размере. В дальнейшем, при необходимости, можно реализовать альтернативный механизм хранения контекста, который не

подвержен данному ограничению, например, большой контекст помещать в файл, а в основном контексте оставлять ссылку на файл.

3.9. Отчеты об ошибках

Были создали два представления для получения информации об популярных ошибках и о самых часто встречающихся.

```
create or replace view errm$most_often_errors as
with
total_count as (
    select count(*) cnt
    from ERRM$LOG
),
error_count as (
    select code, count(*) cnt
    from ERRM$LOG
    group by code
)
select ec.CODE, ec.cnt, e.name, e.TYPE, et.NAME type_name, e.INFO,
e.USAGE_COMMENT
from error_count ec left join ERRM$ERRORS e on ec.CODE = e.CODE left join
ERRM$ERROR_TYPES et on e.TYPE = et.CODE
where cnt > 0.1 * (select cnt from total_count);

create or replace view errm$most_popular_errors as
with
error_count as (
    select code, count(*) cnt
    from ERRM$LOG
    group by code
)
select ec.CODE, ec.cnt, e.name, e.TYPE, et.NAME type_name, e.INFO,
e.USAGE_COMMENT
from error_count ec left join ERRM$ERRORS e on ec.CODE = e.CODE left join
ERRM$ERROR_TYPES et on e.TYPE = et.CODE
order by ec.cnt desc;
```

Рис.3.15. Создание представлений

Код на рисунке рис.3.15, показывает, как создаются указанные представления. В представлении ERRM$MOST_OFTEN_ERRORS содержится информация о тех ошибках, что встречаются чаще 10% от общего числа сохраненных ошибок. Представление ERRM$MOST_POPULAR_ERRORS содержит список всех возникших ошибок, отсортированный по количеству появлений.

3.10. Удаление пакета

Для удаления пакета из системы был разработан скрипт. В котором реализована правильная последовательность удалений объектов пакета.

```
drop trigger errm$additional_info_biu_trg;
drop trigger errm$parameters_biu_trg;
drop trigger errm$feature_flags_biu_trg;

drop table ERRM$PARAMETERS;
drop table ERRM$FEATURE_FLAGS;

drop table ERRM$LOG;
drop table ERRM$ERRORS;
drop table ERRM$ERROR_TYPES;

drop sequence ERRM$ERRORS_CODE_SEQ;
drop sequence ERRM$ERRORS_TYPE_SEQ;
drop sequence ERRM$LOG_SEQ;

drop package ERRM;

drop directory errm$log_dir;
```

Рис.3.16. Скрипт удаления пакета

На рисунке рис.3.16 представлен данный скрипт. В первую очередь удалим триггеры. Далее необходимо удалить объекты в обратной последовательности от их создания. Сначала удаляются таблицы, не имеющие зависимостей, затем происходит удаление таблиц, на которые не ссылаются другие таблицы (такая таблица до удаления единственная и это ERRM$LOG, поэтому она удаляется первой, затем таблица ERRM$ERRORS не имеет таблиц, ссылающихся на нее, и может быть удалена), и далее по цепочке. Затем удаляются созданные во время установки последовательности. В последнюю очередь удаляется сам пакет и директория для логирования. Полный код пакета, и всех остальных скриптов можно посмотреть в приложении 1.

3.11. Выводы

В данной главе был разработан пакет для работы с ошибками, были созданы скрипты для установки и удаления данного пакета, разобраны основные методы пакета. Рассмотрены алгоритмы сохранения информации об ошибках.

ГЛАВА 4. ТЕСТИРОВАНИЕ И АПРОБАЦИЯ ПАКЕТА

Хорошим стилем является наличие введения к главе. Во введении может быть описана цель написания главы, а также приведена краткая структура главы.

4.1. Проверка установки пакета

В первую очередь была написана инструкция по установке и удалению, и помещена к инсталлеру. После чего был протестирован скрипт создания пакета и его удаления. Замеченные недостатки устранялись, на данном этапе была поправлена последовательность создания и удаления объектов. Была проверена установка значений параметров по умолчанию, в ходе тестирования были модифицированы некоторые значения параметров, для максимального быстродействия.

4.2. Проверка работы пакета

Для тестирования пакета был написан скрипт, содержащий тестовые случаи, который был добавлен к скрипту установщику. Была проверена работа всех, объявленных в спецификации пакета, процедур и функций в различных ситуациях и с разными настройками пакета. Рассматривались как положительные случаи, так и отрицательные ситуации (некорректные данные, отсутствующие данные). Часть тестов была автоматизирована, часть оставлена для ручной проверки. Автоматические тесты выполняют несколько ситуаций и сравнивают результат с эталоном, после чего выводят на экран результат проверки.

На рисунке рис.4.1 представлена часть кода автоматических тестов.

```
declare
    l_bool_value boolean;
    l_varchar_value varchar2(2000);
    l_int_value pls_integer;
begin
--Auto test
--No data found
    errm.LOG('Test 1');
    begin
        errm.RAISE('code_implemented');
    exception
        when no_data_found then
            errm.LOG('Correct!');
        when others then
            errm.LOG('Error!');
    end;

    errm.LOG('Test 2');
    begin
        l_bool_value := errm.GET_FEATURE_VALUE('unknown_feature');
    exception
        when no_data_found then
            errm.LOG('Correct!');
        when others then
            errm.LOG('Error!');
    end;

...
```

Рис.4.1. Пример кода автоматических тестов

Первый тест проверяет вызов несуществующий ошибки по ее имени. Второй, пытается получить значение опции, по имени, которое не содержится в таблице ERRM$FEATURE_FLAGS.

Остальные автоматические тесты проверяют следующие ситуации.

Тест 3 устанавливает значение несуществующей опции.

Тест 4 устанавливает значение параметра, которого не существует.

Тест 5 пытается получить значение отсутствующего параметра.

В тесте 6 проверяется ситуация получения кода ошибки, которая не определена.

Вызов ошибки с несуществующим кодом проводится в тесте 7.

Тест 8 повторяет 7 тест, но запрещает логирование.

Тест 9 вызывает ошибку по имени, которой не существует.

Тест 10 делает тоже самое, но без логирования.

Тесты 11 и 12 проверяют методы SRAISE с кодом и именем не существующих ошибок, соответственно. Каждый тест обернут в блок beginend с обработчиком ошибки, если возникла правильная ошибка, тест возвращает корректный результат, если возникла другая ошибка, тест сообщает о неудаче.

9 ручных тестов предназначены для проверки корректности работы основных методов, а также некоторых параметров и опций пакета.

```
--Manual test
--Context added to error
    errm.ADD_DEFAULT_CONTEXT;
    errm.ADD_CONTEXT('user_context', 'user_value');
    errm.DEFAULT_ERROR;

--No context
    errm.SET_FEATURE_VALUE(errm.FEATURE_ADD_DEFAULT_CONTEXT, false);
    errm.ADD_CONTEXT('user_context', 'user_value');
    errm.CLEAR_CONTEXT;
    errm.DEFAULT_ERROR;

--Default context
    errm.SET_FEATURE_VALUE(errm.FEATURE_ADD_DEFAULT_CONTEXT, true);
    errm.ADD_CONTEXT('user_context', 'user_value');
    errm.CLEAR_CONTEXT;
    errm.DEFAULT_ERROR;

--Table ERRM$LOG should be clean
    errm.CLEAR_LOG_TABLE;

--Table ERRM$LOG should be clean
    errm.SET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_TABLE, false);
    errm.DEFAULT_ERROR;

--Correct types for each error
    errm.DEFAULT_ERROR;
    errm.DEFAULT_LOG;
    errm.DEFAULT_WARNING;
    errm.NOT_IMPLEMENTED_EXCEPTION;
    errm.CRITICAL_ERROR;

--Error occurred
    errm.raise('critical_error');
    errm.RAISE(4);

--Error occurred with no additional log
    errm.SET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_TABLE, true);
    errm.SRAISE(4);

--Added information to file
    errm.SET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_FILE, true);
    errm.DEFAULT_LOG;
```

Рис.4.2. Код для ручного тестирования пакета

На рисунке рис.4.2 представлены ручные тесты. Каждый случай содержит комментарий с описанием ожидаемого результата.

Первая ситуация проверяет добавление контекста к ошибке.

Второй тест проверяет возможность очистки контекста.

Третий тест демонстрирует как должна работать очистка контекста, в ситуации с автоматическим добавлением стандартного контекста.

Четвертый тест проверяет корректность очистки таблицы логирования.

Тест под номер пять проверяет флаг для отключения логирования в таблицу.

Тест 6 состоит из вызова нескольких предопределенных ошибок, здесь проверяется корректность типов ошибок, и в целом работа всей системы.

Тест 7 проверяет вызов ошибки по имени и коду.

Тест 8 проверяет корректность работы методов «тихого» вызова исключения.

Тест 9 проверяет запись в файл логирования.

Помимо этого, была проверена запись логов в файл, корректность отработки внутренних (скрытых от пользователя в теле пакета) методов, были проверены ситуации добавления значений в контекст с одинаковым ключом, множественное добавление базового контекста и некоторые другие случаи.

Все найденные в ходе тестирования ошибки и недочеты исправлялись.

Не удалось настроить работу почтового сервиса на виртуальной машине, поэтому тестирование отправки Email сообщение провалилось. Данная опция была отключена по умолчанию.

4.3. Улучшение кода

Во время проверки составлялся список субъективных ощущений от работы с пакетом, после чего данный список был проанализирован, неудобства в работе с пакетом скорректированы.

Был проведен рефакторинг с целью улучшения работы пакета. В ходе него были удалены неиспользуемые переменные и методы, были устранены некоторые узкие места, были пересмотрены используемые типы данных, и сигнатуры некоторых методов.

После рефакторинга было проведено дополнительно полное тестирование пакета, с целью выявление регресса. Найденные проблемы были устранены.

4.4. Сравнение полученных результатов

На рисунке рис.4.3 представлены три примера кода, работающего с исключительными ситуациями.

```
--Example 1
declare
        not_implemented_exception EXCEPTION;
        pragma exception_init(not_implemented_exception, -20000);
begin
        raise not_implemented_exception;
end;
/

--Example 2
begin
        errm.register_error(p_name => 'new_not_implemented', p_info => 'Code not implemented');
        errm.raise('new_not_implemented');
end;
/

--Example 3
begin
        errm.not_implemented_exception;
end;
/
```

Рис.4.3. Примеры использования пакета

Пример 1 использует стандартные средства для вызова исключения. Пример 2 объявляет новое исключение при помощи пакета, а затем вызывает его. Пример 3 вызывает существующее исключение.

Второй код больше приближен к первому примеру, так как создается новое исключение, и выигрыш в объеме кода является не таким значимым, первый случай содержит 150 символов, в то время как во второй ситуации только 136, но второй пример имеет значительный выигрыш в других показателях.

Результаты выполнения первого примера показаны на рисунке рис.4.4:

```
Error report -
ORA-20000:
ORA-06512: на  line 5
20000. 00000 -  "%s"
*Cause:    The stored procedure 'raise_application_error'
           was called which causes this error to be generated.
*Action:   Correct the problem as described in the error message or contact
           the application administrator or DBA for more information.
```

Рис.4.4. Результаты выполнения первого примера

На рисунке рис.4.5 представлены результаты выполнения второго кода:

```
ORA-20000: new_not_implemented (#5)
Code not implemented
Type: error
Context:
    current_schema: DIPLOMA
    db_name: orcl
    host: DESKTOP-E060029
    instance: orcl
    module: SQL Developer
    user_name: DIPLOMA

ORA-06512: на  "DIPLOMA.ERRM", line 292
ORA-06512: на  "DIPLOMA.ERRM", line 300
ORA-06512: на  line 3
20000. 00000 -  "%s"
*Cause:    The stored procedure 'raise_application_error'
           was called which causes this error to be generated.
*Action:   Correct the problem as described in the error message or contact
           the application administrator or DBA for more information.
```

Рис.4.5. Результаты выполнения второго примера

В первом случае мы не получили никакой информации об ошибке, сообщение пустое, почему возникло исключение в коде не понятно. В то время второй код, содержит описание об возникшей ошибке (которая только что создана), а также дополнительную мета информацию, помимо этого сообщение об ошибке будет занесено в таблицу логирования, и в файл логов. Созданная ошибка сразу будет видна другим исполняемым блокам, в то же время информация об ошибке в первом примере доступна только самому анонимному блоку и его вложенным блокам.

Созданную ошибку можно заново использовать без дополнительного кода. Нет необходимости следить за кодами ошибок. Информацию обо всех существующих ошибках можно легко узнать из одной таблицы ERRM$ERRORS.

Пример под номером три, показывает работу пакета в более реальной ситуации, обычно мы хотим использовать уже известную исключительную ситуацию, для часто используемых ошибок, достаточно вызвать единственный метод без параметров. При этом мы получим результат подобный примеру 2, и соответственно все его преимущества. Третий пример состоит всего из 44 символов, следовательно мы потратим меньше времени на написание кода, а важнее то, что нам не потребуется выполнять дополнительные действия для простых ситуаций.

Следующий пример, представленный на рисунке рис.4.6, демонстрирует использование контекста для сохранения значений параметров.

```
begin
        errm.add_context('local_variable', 'null');
        errm.default_error;
end;
/
```

Рис.4.6. Пример добавления контекста

В результате выполнения этого скрипта, в сообщение об ошибке (а также во всех местах куда логируется информация, согласно настройкам) будет содержаться указанная нами информация, что в перспективе облегчит отладку, и исправление ошибки.

Указанные выше примеры, а также некоторые дополнительные, были объединены в один скрипт, который будет распространяться вместе с пакетом для демонстрации различных возможностей приложения.

4.5. Выводы

В данной главе было проведено тестирование кода, а также его улучшение и настройка. Были приведены примеры работы с пакетом, а также сравнение разработанного пакета в работе со стандартным способом.

ЗАКЛЮЧЕНИЕ

В ходе данной работы было предложено решение для автоматизации и систематизации обработки ошибок в Oracle PL/SQL. Согласно с выдвинутыми методами, был разработан пакет для взаимодействия с исключительными ситуациями. Работа реализованного пакета была тщательно протестирована, было проведено сравнение полученного решения со стандартными методами обработки ошибок.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация к PostgreSQL 13.2. — URL: https://postgrespro.ru/media/docs/postgresql/13/ru/postgres-A4.pdf.

2. *Том К.* Oracle для профессионалов. — СПБ: ДиаСофтЮП, 2003. — 961 с.

3. *Фейерштейн С. П. Б.* Oracle PL/SQL. Для профессионалов. — 6-е изд. — СПБ: Питер, 2015. — 1024 с.

4. *Хант Э. Т. Д.* Прогаммист-прагматик. Путь от подмастерья к мастеру. — СПБ: Лори, 2007. — 289 с.

5. Best Practice PL/SQL. — URL: https://nyoug.org/Presentations/2006/200611_Steven_Feuerstein_Best_Practice_PLSQL/200611_Steven_Feuerstein_Best_Practice_PLSQL.pdf.

6. Handling PL/SQL Errors. — URL: https://docs.oracle.com/cd/B28359_01/appdev.111/b28370/errors.htm#LNPLS007.

7. MySQL 8.0 Reference Manual. — URL: https://dev.mysql.com/doc/refman/8.0/en/signal.html.

8. Transact-SQL (T-SQL) Reference. — URL: https://docs.microsoft.com/en-us/sql/t-sql/language-elements/raiserror-transact-sql?view=sql-server-2017.

9. UTL_FILE Oracle documentation. — URL: https://docs.oracle.com/cd/B28359_01/appdev.111/b28419/u_file.htm#BABGGEDF.

Приложение 1

Код пакета

П1.1. Код установочного скрипта

Файл install_script.sql:

```
--CREATE ERROR_TYPES TABLE
create table errm$error_types (
code number(5) primary key,
name varchar2(100 char) unique not null,
info varchar2(2000 char)
);

create sequence errm$errors_type_seq start with 4 increment by 1;

insert into errm$error_types values (0, 'error', 'Exception');
insert into errm$error_types values (1, 'warning', 'Warning');
insert into errm$error_types values (2, 'log', 'Log information');
insert into errm$error_types values (3, 'critical_error', 'Exceptions th

--CREATE ERRORS TABLE
create table errm$errors (
code number(10) primary key,
name varchar2(100) unique not null,
type number(5) default 0 references errm$error_types not null,
info varchar2(2000) not null,
usage_comment varchar2(1000),
create_date date,
create_by varchar2(100)
);
/

create sequence errm$errors_code_seq start with 5 increment by 1;
```

```
create or replace trigger errm$additional_info_biu_trg
before insert or update
on errm$errors
for each row
begin
:new.name := lower(:new.name);
if :old.create_date is null then
:new.create_date := sysdate;
:new.create_by := SYS_CONTEXT ('USERENV', 'SESSION_USER');
end if;
end errm$additional_info_biu_trg;

insert into errm$errors values (0, 'default_error', 0, 'Error has occurr
insert into errm$errors values (1, 'not_implemented', 0, 'Code not imple
insert into errm$errors values (2, 'default_warning', 1, 'Warning inform
insert into errm$errors values (3, 'default_log', 2, 'Log information',
insert into errm$errors values (4, 'critical_error', 3, 'Critical error

--CREATE LOG TABLE
create table errm$log (
id number(38) primary key,
code number(10) references errm$errors not null,
timestamp timestamp not null,
message varchar2(1000),
context varchar2(2000)
);

create sequence errm$log_seq start with 1 increment by 1;

create table errm$parameters (
name varchar2(50) primary key,
value varchar2(1000)
);

insert into errm$parameters values ('MAIL', 'mikl181@yandex.ru');
```

```
create or replace trigger errm$parameters_biu_trg
before insert or update
on errm$parameters
for each row
begin
:NEW.name := upper(:NEW.name);
:NEW.value := lower(:NEW.value);
end errm$parameters_biu_trg;

create table errm$feature_flags (
name varchar2(50) primary key,
value char(1) default '0'
);

insert into errm$feature_flags values ('ALWAYS_ADD_DEFAULT_CONTEXT', '1'
insert into errm$feature_flags values ('MAIL_CRITICAL_ERRORS', '0');
insert into errm$feature_flags values ('LOG_ERRORS_TO_FILE', '1');
insert into errm$feature_flags values ('LOG_ERRORS_TO_TABLE', '1');

create or replace trigger errm$feature_flags_biu_trg
before insert or update
on errm$feature_flags
for each row
begin
:NEW.name := upper(:NEW.name);

if (:new.value not in ('0', '1')) then
raise_application_error(-20000, 'Invalid data');
end if;
end errm$feature_flags_biu_trg;

create or replace view errm$most_often_errors as
with
total_count as (
```

```
select count(*) cnt
from ERRM$LOG
),
error_count as (
select code, count(*) cnt
from ERRM$LOG
group by code
)
select ec.CODE, ec.cnt, e.name, e.TYPE, et.NAME type_name, e.INFO, e.USA
from error_count ec left join ERRM$ERRORS e on ec.CODE = e.CODE left joi
where cnt > 0.1 * (select cnt from total_count);

create or replace view errm$most_popular_errors as
with
error_count as (
select code, count(*) cnt
from ERRM$LOG
group by code
)
select ec.CODE, ec.cnt, e.name, e.TYPE, et.NAME type_name, e.INFO, e.USA
from error_count ec left join ERRM$ERRORS e on ec.CODE = e.CODE left joi
order by ec.cnt desc;

create directory errm$log_dir as 'E:/ERRM';

@@errm.pks
@@errm.pkb
```

П1.2. Код спецификации пакета

Файл errm.pks:

```
create or replace package errm
is
--CONSTANTS
```

```
feature_mail_critical_errors constant varchar2(50 char) := 'MAIL_CRITICA
feature_log_errors_to_file constant varchar2(50 char) := 'LOG_ERRORS_TO_
feature_log_errors_to_table constant varchar2(50 char) := 'LOG_ERRORS_TO
feature_add_default_context constant varchar2(50 char) := 'ALWAYS_ADD_DE

parameter_mail constant varchar2(50 char) := 'MAIL';

--TYPES
subtype t_err_code is pls_integer;
subtype t_err_name is varchar2(100 char);
subtype t_err_type is pls_integer;

--PARAMETERS WORK
function get_feature_value(p_feature_name in varchar2) return boolean;
procedure set_feature_value(p_feature_name in varchar2, p_value in BOOLE

function get_parameter_value(p_param_name in varchar2) return varchar2;
procedure set_parameter_value(p_param_name in varchar2, p_value in varch

--HELPERS
function get_error_code(p_name in t_err_name) return t_err_code;

--PROCEDURE AND FUNCTIONS
procedure register_error(p_code in t_err_code := null, p_name in t_err_n

procedure add_default_context;
procedure add_context(p_name in varchar2, p_value in varchar2);
procedure clear_context;

procedure raise(p_code in t_err_code, p_log_enabled in boolean := TRUE);
procedure raise(p_name in t_err_name, p_log_enabled in boolean := TRUE);

--SILENT RAISES
procedure sraise(p_code in t_err_code);
procedure sraise(p_name in t_err_name);
```

```
--LOG WORK
procedure clear_log_table;
procedure clear_log_file;

--SIMPLE ERRORS RAISE
procedure default_error;
procedure default_log;
procedure default_warning;
procedure critical_error;
procedure not_implemented_exception;

--LOG
procedure log (p_text in varchar2);
procedure nl;
end errm;
/
```

П1.3. Код тела пакета

Файл errm.pkb:

```
create or replace package body errm
is
--CONSTANTS

c_default_error constant pls_integer := -20000;
--New line symbol
c_nl constant char(1) := chr(10);


--TYPES

TYPE t_context IS TABLE OF VARCHAR2(100) INDEX BY VARCHAR2(100);
```

```
--VARIABLES

g_context t_context;

--PARAMETERS WORK

function get_feature_value(p_feature_name varchar2)
return boolean
is
l_value errm$feature_flags.value%type;
begin
select value
into l_value
from ERRM$FEATURE_FLAGS
where NAME = upper(p_feature_name);

return l_value = '1';
end get_feature_value;

procedure set_feature_value(p_feature_name varchar2, p_value BOOLEAN)
is
pragma autonomous_transaction;
begin

if p_value then
update ERRM$FEATURE_FLAGS
set value = '1'
where NAME = upper(p_feature_name);
else
update ERRM$FEATURE_FLAGS
set value = '0'
where NAME = upper(p_feature_name);
end if;

if sql%rowcount = 0 then
```

```
raise no_data_found;
end if;
commit;
end set_feature_value;

function get_parameter_value(p_param_name varchar2)
return varchar2
is
l_value ERRM$PARAMETERS.value%type;
begin
select value
into l_value
from ERRM$PARAMETERS
where NAME = upper(p_param_name);

return l_value;
end get_parameter_value;

procedure set_parameter_value(p_param_name varchar2, p_value varchar2)
is
pragma autonomous_transaction;
begin
update ERRM$PARAMETERS
set value = lower(p_value)
where NAME = upper(p_param_name);

if sql%rowcount = 0 then
raise no_data_found;
end if;
commit;
end set_parameter_value;


--HELPERS
```

```
procedure reset_seq(p_seq_name in varchar2)
is
pragma autonomous_transaction;
l_val number;
begin
--Hack from AskTom for reset sequence
execute immediate 'select ' || p_seq_name || '.nextval from dual' INTO l
execute immediate 'alter sequence ' || p_seq_name || ' increment by -' |
execute immediate 'select ' || p_seq_name || '.nextval from dual' INTO l
execute immediate 'alter sequence ' || p_seq_name || ' increment by 1 mi
end reset_seq;

procedure add_default_context
is
begin
add_context('instance', SYS_CONTEXT ('USERENV', 'INSTANCE_NAME'));
add_context('db_name', SYS_CONTEXT ('USERENV', 'DB_NAME'));
add_context('user_name', SYS_CONTEXT ('USERENV', 'SESSION_USER'));
add_context('module', SYS_CONTEXT ('USERENV', 'MODULE'));
add_context('current_schema', SYS_CONTEXT ('USERENV', 'CURRENT_SCHEMA'))
add_context('host', SYS_CONTEXT ('USERENV', 'HOST'));
end add_default_context;

procedure add_context(p_name in varchar2, p_value in varchar2)
is
begin
g_context(p_name) := p_value;
end add_context;

procedure clear_context
is
begin
g_context.DELETE();
end clear_context;
```

```
function get_error_info(p_code in t_err_code)
return errm$errors%rowtype
is
l_error errm$errors%rowtype;
begin
select *
into l_error
from ERRM$ERRORS
where code = p_code;

return l_error;
end get_error_info;

function get_context_string(p_use_tab boolean := true)
return varchar2
is
l_index   varchar2(100);
l_context varchar2(1000);
begin
if get_feature_value(errm.FEATURE_ADD_DEFAULT_CONTEXT) then
add_default_context();
end if;

if g_context.COUNT = 0 then
return null;
end if;

l_index := g_context.FIRST;

while l_index is not null
loop
l_context := l_context || case when p_use_tab then '    ' else '' end ||
l_index := g_context.next(l_index);
end loop;
```

```
return l_context;
end get_context_string;

function get_error_message(p_code in t_err_code)
return varchar2
is
r_error_instance errm$errors%rowtype;
l_error_typename errm$error_types.name%type;
l_context varchar2(2000);
begin
r_error_instance := get_error_info(p_code);

select name
into l_error_typename
from ERRM$ERROR_TYPES
where code = r_error_instance.TYPE;

l_context := get_context_string();

return  r_error_instance.name || ' (#' || p_code ||')' || c_nl ||
r_error_instance.info || c_nl ||
'Type: ' || l_error_typename || c_nl ||
'Context' ||
case
when l_context is null then
' not provided!'
else
': ' || c_nl || l_context
end
|| c_nl;
end get_error_message;


procedure log_to_file(p_code in t_err_code)
is
```

```
l_log UTL_FILE.file_type;
begin
if NOT get_feature_value(FEATURE_LOG_ERRORS_TO_FILE) then
return;
end if;

l_log := UTL_FILE.FOPEN('ERRM$LOG_DIR', 'log.txt','A'); -- A - append
UTL_FILE.PUT(l_log, systimestamp || ': ' || get_error_message(p_code));
UTL_FILE.FCLOSE(l_log);
end log_to_file;

procedure log_to_table(p_code in t_err_code)
is
pragma autonomous_transaction;

l_error errm$errors%rowtype;
l_message errm$log.message%type;
l_context errm$log.context%type;
begin
if NOT get_feature_value(FEATURE_LOG_ERRORS_TO_TABLE) then
return;
end if;

l_error := get_error_info(p_code);
l_message := get_error_message(p_code);
l_context := get_context_string(false);

insert into ERRM$LOG values (errm$log_seq.nextval, p_code, systimestamp,
commit;
end log_to_table;

procedure log_to_mail(p_code in t_err_code)
is
l_host varchar2(100);
l_db_name varchar2(100);
```

```
l_error_instance errm$errors%rowtype;
l_message varchar2(2000);
begin
if NOT get_feature_value(FEATURE_MAIL_CRITICAL_ERRORS) then
return;
end if;
l_error_instance := get_error_info(p_code);

if l_error_instance.TYPE != 4 then
return;
end if;

l_host := SYS_CONTEXT ('USERENV', 'HOST');
l_db_name := SYS_CONTEXT ('USERENV', 'DB_NAME');
l_message := get_error_message(p_code);

UTL_MAIL.send(
sender     => l_db_name || '@' || l_host || '.com',
recipients => get_parameter_value(errm.PARAMETER_MAIL),
subject    => l_error_instance.NAME || ' has occurred on ' || l_db_name
message    => l_message);
end log_to_mail;

procedure log_error(p_code in t_err_code)
is
begin
log_to_table(p_code);
log_to_file(p_code);
log_to_mail(p_code);
end log_error;

function get_error_code(p_name in t_err_name)
return t_err_code
is
l_err_code t_err_code;
```

```
begin
select code
into l_err_code
from errm$errors
where lower(name) = lower(p_name);

return l_err_code;
end get_error_code;


--PROCEDURE AND FUNCTIONS

procedure register_error(p_code in t_err_code := null, p_name in t_err_n
is
pragma autonomous_transaction;
begin
insert into ERRM$ERRORS
values (NVL(p_code, errm$errors_code_seq.nextval), lower(p_name), p_type
commit;
end register_error;


procedure raise(p_code in t_err_code, p_log_enabled in boolean := TRUE)
is
l_message varchar2(2000);
begin
if p_log_enabled then
log_error(p_code);
end if;

l_message := get_error_message(p_code);

clear_context();
raise_application_error(c_default_error, l_message);
end raise;
```

```
procedure raise(p_name in t_err_name, p_log_enabled in boolean := TRUE)
is
l_err_code t_err_code;
begin
l_err_code := get_error_code(p_name);
errm.raise(l_err_code, p_log_enabled);
end raise;

procedure sraise(p_code in t_err_code)
is
begin
errm.raise(p_code, FALSE);
end sraise;

procedure sraise(p_name in t_err_name)
is
begin
errm.raise(p_name, FALSE);
end sraise;


--LOG WORK

procedure clear_log_table
is
pragma autonomous_transaction;
begin
execute immediate 'truncate table errm$log';

--We can't drop and recreate our sequence, because package should be com
reset_seq('ERRM$LOG_SEQ');
end clear_log_table;

procedure clear_log_file
```

```
is
l_log UTL_FILE.file_type;
begin
l_log := UTL_FILE.FOPEN('ERRM$LOG_DIR', 'log.txt','W');
UTL_FILE.FCLOSE(l_log);
end clear_log_file;

--EASE ERRORS RAISE

procedure default_error
is
begin
errm.raise(0);
end default_error;

procedure not_implemented_exception
is
begin
errm.raise(1);
end not_implemented_exception;

procedure default_warning
is
begin
errm.raise(2);
end default_warning;

procedure default_log
is
begin
errm.raise(3);
end default_log;

procedure critical_error
is
```

```
begin
errm.raise(4);
end critical_error;


--LOG

procedure log (p_text in varchar2)
is
begin
dbms_output.put_line(p_text);
end log;

procedure nl
is
begin
dbms_output.new_line;
end nl;
end errm;
/
```

П1.4. Код скрипта для удаления

Файл uninstall.sql:

```
drop view ERRM$MOST_OFTEN_ERRORS;
drop view ERRM$MOST_POPULAR_ERRORS;

drop trigger errm$additional_info_biu_trg;
drop trigger errm$parameters_biu_trg;
drop trigger errm$feature_flags_biu_trg;

drop table ERRM$PARAMETERS;
drop table ERRM$FEATURE_FLAGS;
```

```
drop table ERRM$LOG;
drop table ERRM$ERRORS;
drop table ERRM$ERROR_TYPES;

drop sequence ERRM$ERRORS_CODE_SEQ;
drop sequence ERRM$ERRORS_TYPE_SEQ;
drop sequence ERRM$LOG_SEQ;

drop package ERRM;

drop directory errm$log_dir;
```

П1.5. Файл с инструкциями по установке

Файл readme.txt:

```
Инструкция по установке пакета.
1. Найти в файле install.sql строку
create directory errm$log_dir as 'E:/ERRM';
2. Заменить путь директории на необходимый для хранения фалйов логирован
3. Настроить пакет UTL_MAIL
4. Выполнить скрипт install.sql
5. Прочитать описание параметров и опций
6. Настроить пакет по своему усмотрению при помощи процедур SET_PARAMETE

Удаление пакета:
1. Запустить скрипт uninstall.sql
```

П1.6. Скрипт с примерами использования пакета

Файл examples.sql:

```
--Example 1. Standard exception raise
declare
not_implemented_exception EXCEPTION;
pragma exception_init(not_implemented_exception, -20000);
```

```
begin
raise not_implemented_exception;
end;
/

--Example 2. Define new error and raise
begin
errm.register_error(p_name => 'new_not_implemented', p_info => 'Code not
errm.raise('new_not_implemented');
end;
/

--Example 3. Raise predefined error
begin
errm.not_implemented_exception;
end;
/

--Example 4. Using context for save information
begin
errm.add_context('local_variable', 'null');
errm.default_error;
end;
/

--Example 5. Silent raise with code
begin
errm.sraise(1);
end;
/

--Example 6. Silent raise with name
begin
errm.sraise('critical_error');
end;
```

```
/

--Example 7. Raise with name
begin
errm.raise('critical_error');
end;
/

--Example 8. Raise with error code
begin
errm.raise(0);
end;
/
```

П1.7. Скрипт для тестирования пакета

Файл test.sql:

```
SET SERVEROUTPUT ON;
declare
l_bool_value boolean;
l_varchar_value varchar2(2000);
l_int_value pls_integer;
begin
--Auto test
--No data found
errm.LOG('Test 1');
begin
errm.RAISE('code_implemented');
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;
```

```
errm.LOG('Test 2');
begin
l_bool_value := errm.GET_FEATURE_VALUE('unknown_feature');
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 3');
begin
errm.SET_FEATURE_VALUE('unknown_feature', false);
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 4');
begin
l_varchar_value := errm.GET_PARAMETER_VALUE('unknown_parameter');
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 5');
begin
errm.SET_PARAMETER_VALUE('unknown_parameter', 'unknown_value');
exception
```

```
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 6');
begin
l_int_value := errm.GET_ERROR_CODE('unknown_error');
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 7');
begin
errm.RAISE(-1);
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 8');
begin
errm.RAISE(-1, false);
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;
```

```
errm.LOG('Test 9');
begin
errm.RAISE('unknown_error');
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 10');
begin
errm.RAISE('unknown_error', false);
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 11');
begin
errm.SRAISE(-1);
exception
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

errm.LOG('Test 12');
begin
errm.SRAISE('unknown_error');
exception
```

```
when no_data_found then
errm.LOG('Correct!');
when others then
errm.LOG('Error!');
end;

--Manual test
--Context added to error
--     errm.ADD_DEFAULT_CONTEXT;
--     errm.ADD_CONTEXT('user_context', 'user_value');
--     errm.DEFAULT_ERROR;

--No context
--     errm.SET_FEATURE_VALUE(errm.FEATURE_ADD_DEFAULT_CONTEXT, false);
--     errm.ADD_CONTEXT('user_context', 'user_value');
--     errm.CLEAR_CONTEXT;
--     errm.DEFAULT_ERROR;

--Default context
--     errm.SET_FEATURE_VALUE(errm.FEATURE_ADD_DEFAULT_CONTEXT, true);
--     errm.ADD_CONTEXT('user_context', 'user_value');
--     errm.CLEAR_CONTEXT;
--     errm.DEFAULT_ERROR;

--Table ERRM$LOG should be clean
--     errm.CLEAR_LOG_TABLE;

--Table ERRM$LOG should be clean
--     errm.SET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_TABLE, false);
--     errm.DEFAULT_ERROR;

--Correct types for each error
--     errm.DEFAULT_ERROR;
--     errm.DEFAULT_LOG;
--     errm.DEFAULT_WARNING;
```

```
--      errm.NOT_IMPLEMENTED_EXCEPTION;
--      errm.CRITICAL_ERROR;

--Error occurred
--      errm.raise('critical_error');
--      errm.RAISE(4);

--Error occurred with no additional log
--      errm.SET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_TABLE, true);
--      errm.SRAISE(4);

--Added information to file
--      errm.SET_FEATURE_VALUE(errm.FEATURE_LOG_ERRORS_TO_FILE, true);
--      errm.DEFAULT_LOG;
end;
/
```