# Этапы разработки проекта: заключительные стадии проектирования, спецификации функций

## Обработка иерархии функций

На этапе анализа была создана иерархия функций, которая дополняется диаграммами потока данных и изменения состояний. Как проще всего проверить, является ли функция в иерархии атомарной? Ответьте на вопрос: имеет ли смысл выполнение только части этой функции. Поскольку мы имеем дело с иерархией, вероятно наличие одной и той же функции в нескольких местах иерархии. Здесь кроется ошибка анализа: аналитик перепутал функцию и механизм.

Хорошо, если при описании функции аналитик укажет в скобках тот объект в ER-модели (сущность и атрибут), о котором идет речь, например «выбрать товары в накладной из списка товаров (ITEMS), хранящихся на складе (STORE)». Если подобных указаний нет, это станет хорошим упражнением для приемки отчетов аналитиков; заодно и убедитесь, что вы правильно понимаете друг друга. Параллельно проверьте, есть ли сущности, которые не используются ни в одной функции, - это поможет сделать матрицу «функции-атрибуты», отражающую факт использования в функциях атрибутов сущностей.

Как правило, не бывает взаимно однозначного отображения функций на модули. Если однозначных соответствий много, то вероятнее всего был выполнен не анализ, а собственно проектирование. Зачастую схожие по выполняемым действиям функции объединяют, даже если у них разный контекст. Некоторые сложные функции разбивают на более простые модули. Некоторые функции преобразуют в ограничения базы данных (constraints, или триггеры и хранимые процедуры). Каких-то общих способов отображения функций на модули не существует. Требуется время, терпение, опыт. Проектировщики часто меняют количество, состав модулей в течение процесса проектирования, и это правило, а не исключение.

Некоторые группы модулей присутствуют в любом проекте:

* работа с базой данных:
* обработка кодов возврата СУБД;
* установка соединения с СУБД и его параметры;
* выполнение некурсорного запроса (DDL- и DML-операции);
* выполнение выборки;
* вызов хранимой процедуры и обработка ее ответа;
* операции начала и завершения транзакции;
* функциональные модули:
* модули, реализующие бизнес-процессы;
* модули, изолирующие обработку запросов к базе данных и интерфейсы, предоставляемые пользователю;
* системные модули:
* средства управления приложениями;
* средства обмена данными с внешними системами;
* планировщик пакетных заданий;
* менеджер печати документов;
* модуль резервного копирования;
* модуль архивирования;
* модуль автоматического восстановления при сбоях;
* средства администрирования пользователей системы (регистрация, назначение прав, внешняя аутентификация);
* средства создания нерегламентированных запросов к базе данных.

Один из приемов построения расширяемых систем состоит в создании независимости двух слоев: обработки запросов и интерфейса, предоставляемого пользователю. Можно специально кодировать каждый запрос данных, например именем функции и номером или как-нибудь иначе. Текст SQL-запроса скрыт от разработчиков интерфейсов; они имеют доступ только к возвращаемому запросом множеству - коду ошибки или выборке. Формат кода ошибки и выборки фиксируется. Это позволяет проектировщикам схемы базы данных изменять как ее, так и сам SQL-запрос, однако эти изменения не отражаются на процессах создания интерфейсов. Аналогичная независимость реализуется для слоя функций, обеспечивающих вызовы интерфейса, предоставляемого СУБД для выполнения запросов. Этот слой функций может использовать вызовы как native-интерфейса СУБД, так и стандартных интерфейсов, например ODBC.

Очень важен слой функций обработки ошибок. При выполнении запросов к базе данных всегда следует предусматривать обработку исключений, например нарушений ограничений целостности. Отдельно требуется предусмотреть обработку конфликтов транзакций и принудительный откат текущей транзакции вследствие разрешения конфликтов типа deadlock. Проектировщики должны хорошо понимать особенности многопользовательской работы, которые реализованы используемой в проекте СУБД. Проектирование потоков транзакций и снижение конфликтов между ними - одна из самых сложных задач проектирования.

Для больших проектов важно наличие функций мониторинга работы приложений. Они служат в том числе и для оценки корректности работы самих приложений, а также для сбора статистики отказов системы и прогнозирования вероятных отказов. Такие функции отслеживают моменты перегрузки системы и автоматически разрешают их, уведомляя администратора о принятом решении. Подобные модули могут быть реализованы в виде локальных агентов, каждый из которых работает на одном сервере, а сами они обмениваются информацией друг с другом и центральной программой управления.

## Управление исходным кодом

Для групповой разработки важны системы контроля исходного кода. Такие системы решают по крайней мере две задачи: хранение всех версий каждого экземпляра исходного кода (версии файлов) и разрешение конфликтов одновременного доступа разработчиков к одному экземпляру исходного кода (слияние исходных текстов, согласованность группы файлов проекта и т.п.)

Контролю подвергается не только собственно код модулей, но и код скриптов генерации схемы базы данных, а также собственно схема базы данных, которая может храниться в виде версий бинарных файлов, экспортированных во внутренний формат СУБД схем баз данных. Дело проектировщиков - предусмотреть наличие таких механизмов разработки, дело прикладных программистов - придерживаться установленных правил групповой разработки. Часто ведущие исполнители проектов не доверяют системам контроля исходного кода сливать правки в исходных текстах автоматически и частично или полностью контролируют этот процесс. Эта перестраховка во многих случаях себя оправдывает.

После того как основной код модуля будет создан и зарегистрирован в системе контроля исходных текстов, каждая правка в нем должна быть помечена датой и именем автора правки. Правки должны сопровождаться ясными комментариями, которые располагаются в начале файла исходного кода. Также в начале исходного кода (в комментариях) описывается: для чего данный файл исходного кода создан, основные его функции, к какой части информационной системы он относится, кто автор. Формат комментария к правке может быть таким

10.01.2000 Ivanov: authorization bug fixed (found by Petrov)

Это делается для того, чтобы через некоторое время можно было понять, кто и зачем внес данную правку. Также это помогает корректировать слияние исходного кода, если система контроля обнаружила несовместность правок и не может разрешить ее сама.

Функции, структуры, наиболее важные переменные должны сопровождаться комментариями. Избегайте непонятных названий вида K1, Function10.

Для крупных подсистем следует зафиксировать интерфейс обмена с другими подсистемами - формат данных, передаваемых подсистеме и получаемых из нее, а также формат вызовов функций и методов подсистемы. Это позволит добиться относительной независимости подсистем и снизить влияние изменений кода одной подсистемы на другую. Такой подход облегчает взаимодействие разработчиков разных модулей. Документировать придется только интерфейс обмена, а не весь код модуля целиком. Зачастую разработчику, вызывающему функцию модуля, требуется знать, что нужно передать и как обработать результат, а что происходит внутри - знать не обязательно. Единственное, что его интересует, - это правильная инициализация модуля, правильный прием результатов его работы и правильная выгрузка модуля. Взаимодействие компонентов не должно быть произвольным. Контролировать эти правила создания исходного кода в большинстве случаев приходится вручную.

Следует отметить, что группа разработчиков должна иметь свой выделенный сервер баз данных, и, возможно, не один, а также выделенные рабочие места. Часто эти моменты далеко не очевидны руководству, и оно воспринимает это как совершенно ненужную трату средств. Сервер, обеспечивающий контроль исходного кода, также должен быть выделенным.

## Размещение логики обработки

В отчетах аналитиков часто смешиваются три группы правил: правила для данных, процессов и интерфейса. На этапе проектирования эти правила предстоит выделить.

В правилах для данных формулируются условия, которым всегда должны удовлетворять данные. Такие правила безоговорочны; они могут быть неверными, но в этом случае они безоговорочно неверные. Правила данных определяются в схеме базы данных. Они могут контролироваться приложением только в том случае, когда мощности языка данных не хватает для их поддержки (например, поддержка сложных ограничений check, содержащих агрегатные функции, и т.п.).

Правила для процессов определяют, что должно и что не должно делать приложение, и выводятся из модели функций, переданной аналитиками.

В правилах для интерфейсов устанавливается, что должен видеть конечный пользователь. Такие правила не касаются обработки данных, они выводятся из спецификации пользовательского интерфейса (например, к ним относится внешний вид экранных форм отчетов и документов).

Ниже приведем несколько правил, переданных аналитиками, и классифицируем их. Дело в том, что при проектировании группы правил не должны перемешиваться. Итак:

1. Только руководитель может санкционировать выплату вознаграждения.

Это правило только для процессов. В момент разрешения операции приложение должно проверить, есть ли у пользователя привилегии, разрешающие операцию, и нет ли привилегий, запрещающих операцию. Многие проектировщики пытаются реализовать такие правила, как правила данных. Но связь здесь зависит от времени, а не является постоянной (руководитель может быть смещен, подчиненный может стать руководителем, для временных групп работников это процесс постоянный).

1. Обновление записей о платежах запрещено.

Это правило для данных. Оно не изменяется во времени. Такое правило можно реализовать с помощью триггера before update, он должен возвращать ошибку вызвавшему приложению. Откат транзакции внутри триггера в этом случае возможен только при отсутствии сложных транзакций, операции которых инициируют вызов данного триггера.

1. Все коды валют должны раскрываться; рядом с кодом указывается полное название валюты.

Это правило для интерфейса. На первый взгляд оно разумно, но практика показывает, что в большинстве случаев пользователи оперируют кодами, а не полными названиями валют. В большинстве организаций есть система сокращений, ставших частью языка и понятных всем. Такие сокращения не требуют раскрытия.

1. Все торговые операции, выполненные в воскресенье, учитываются в бухгалтерской книге за следующий понедельник.

Подобные правила часто описываются аналитиками. На самом деле здесь два правила в одном, и их надо разделить. Первое гласит о том, что в бухгалтерских книгах не должно быть проводок, сделанных в воскресенье. Это правило данных. Оно может быть реализовано ограничением check. Второе правило - для процессов. Оно объясняет, как откорректировать дату: чтобы дата бухгалтерской проводки была правильной для «воскресенья», требуется прибавить один день. Это позволяет избежать случая, когда в приложении появляется оператор insert с заведомо отвергаемыми данными. Обработку такой ситуации можно предоставить хранимой процедуре, если ее язык достаточно мощный и допустимы вызовы внешних функций (в данном случае проверка дня недели) или имеется встроенная функция обработки календаря. Если же решение в виде хранимой процедуры невозможно, то это преобразование должно выполнить само приложение (вызов библиотечной функции или соответствующая организация объектов - как больше нравится). Другое решение подобной задачи - добавление производного атрибута. Вместо одного атрибута, хранящего дату, получаем два: один хранит реальную дату, второй - откорректированную для «воскресенья»; последний является производным. Это допустимо, если подобная денормализация не влечет за собой много аномалий модификации.

1. Пособие не должно выплачиваться лицу, если его доход превышает 300 руб. на человека.

Это правило для процессов. Здесь не говорится о том, какие данные образуют доход, в разные периоды времени источники дохода разные для одного и того же лица, состав семьи также меняется с течением времени.

Точно такие же группы правил могут быть применены и для проектной документации. Эти группы могут отражать естественное разделение разработчиков на тех, кто реализует работу с базой данных, системные модули и интерфейсы пользователя.

Полное описание интерфейсов на этапе проектирования возможно, если требования пользователя достаточно четко определены. На практике реализация интерфейса пользователя является наиболее часто изменяемой частью исходного кода. На этапе проектирования следует описать наиболее общие правила интерфейса, например горячие клавиши, используемые одинаково во всех программах.

Следует отметить, что много проблем в интерфейсах пользователей создают сами проектировщики, если они неправильно выбирают макет и не учитывают разницы поведения системы на макете и на реальных данных. Наиболее частая ошибка проектирования интерфейса - отображение данных в форме для редактирования и блокирование их средствами СУБД до тех пор, пока пользователь не нажмет кнопку OK. Еще одна распространенная ошибка проектирования интерфейса - обработка длительных процессов, когда пользователь должен ждать ответа на запрос. Большинство проектировщиков не предусматривают единого вызова функции- обработчика события ожидания ответа. На макете запрос может проходить мгновенно, а в случае реальных данных это может составлять несколько минут. Если обработка ожидания ответа не предусмотрена (хотя бы в виде элементарного сообщения «подождите, идет обработка данных»), то пользователь думает, что приложение «зависло».

С точки зрения логики расположения правил они должны быть реализованы так:

* правила интерфейса реализуются во внешней системе, например в Delphi, генераторах экранных форм и отчетов, поставляемых в составе средств разработки СУБД;
* правила для процессов могут быть реализованы как процедуры, вызываемые из внешней системы;
* правила данных следует реализовать средствами СУБД с помощью ограничений целостности.

Следует отметить, что место правил интерфейса и правил данных всегда задано точно. Место правил процессов не всегда определено точно. Часть из них может быть реализована как вызовы утилит, поставляемых с СУБД или созданных другими группами разработчиков; часть может храниться в схеме как процедуры или пакеты; другие могут быть реализованы как библиотечные функции собственно информационной системы. Но любая процедура, реализующая правило процесса, должна быть отделена от кода, реализующего правила интерфейса, и не зависеть от него.

Вопросы проверки корректности ввода информации часто решаются проектировщиками неоднозначно. Как правило, пользователь настаивает на более детальном анализе ошибки ввода данных. Если вводимые значения выбираются из справочника, то здесь проблем меньше. Если они вводятся вручную, то возникает проблема предотвращения попадания в систему некорректных данных, например элементарных опечаток. Выбор значений из справочника - это форма реализации ограничений на уровне интерфейса. Проектировщики часто выбирают - делать проверку на уровне интерфейса или на уровне ограничений базы данных. Лучше реализовать оба правила: и интерфейсное - поскольку пользователь требует немедленной обратной связи, и правило данных - как дополнительную проверку корректности.

При проектировании интерфейсов старайтесь придерживаться стандартов де-факто. Ими могут быть приложения, часто используемые пользователем, например вызов контекстной подсказки по F1 и т.п.

При работе нескольких пользователей с одними и теми же объектами данных проектировщикам приходится решать задачи совместного редактирования документов, например оформление заказа. Клиент не всегда точно определяет список товаров и их количество, а потому оформление заказа может требовать некоторого времени (и оно больше, чем чистое время заполнения экранной формы оператором). Проектировщики допускают распространенную ошибку решения интерфейса пользователя для таких задач: в экранной форме отображается ввод позиций заказа, которые выбираются из справочников; выбранные данные блокируются до тех пор, пока оператор не нажмет OK. Это приводит к возникновению феномена «конвоя». А именно - несколько операторов после нажатия OK начинают ждать разрешения конфликтов средствами СУБД, в то время как большое количество конфликтов спровоцировал именно интерфейс. В самом деле время редактирования формы намного превосходит время обмена данными приложения пользователя и СУБД, соединение с СУБД простаивает до 99% времени - то есть ждет запроса, тогда как блокировка данных остается. Чем дольше время блокировки, тем больше вероятность конфликта. Здесь у некоторых проектировщиков возникает идея обвинить СУБД во всех грехах: она же блокирует, а мы вроде и ни при чем. Предположим, что СУБД не блокирует редактируемые данные, и они вступают в силу только по нажатию кнопки OK. Это, конечно, хорошо, но в течение времени редактирования другой пользователь мог изменить те же данные и зафиксировать свои изменения. СУБД транзакцию второго пользователя уже не пропустит - это типично для стратегии оптимистических блокировок. Возникает вопрос: если и так плохо, и так нехорошо, что делать? Проблема - в неверном взаимодействии интерфейса и обработки данных. Транзакция в любой СУБД начинается или явно, или по факту первого запроса данных. При описанном решении интерфейса транзакция оказывается слишком длинной. Ведь когда идет формирование списка заказываемых товаров, выполняются запросы к данным и транзакция уже начата. Можно принудительно разорвать операцию формирования заказа и его подтверждения. Здесь используют стандартные методы определения зон риска: так, в случае заказа товара это вход в зону потенциальной нехватки (например, заказали 90% товара, имеющегося на складе, - это следует отметить как сигнал потенциального риска продажи товара два раза). По кнопке OK выполняется подтверждение ранее зарезервированного товара и в результате вероятность конфликта снижается.

Аналогично решается задача одновременного редактирования двумя пользователями одного документа. Изменения пользователя (которые он сделал в экранной форме) запоминаются в специальном буфере данных; то, что он первоначально получил для редактирования, также запоминается в буфере начальных данных. По нажатию кнопки OK выполняется попытка зафиксировать изменения пользователя. На уровне правил данных (разрешение конфликтов транзакций) выполняется проверка отсутствия зафиксированных другим пользователем изменений. Если данные были кем-то изменены во время редактирования, то пользователь получает предупреждение об этом. Ему может быть предложено просмотреть изменения, которые находятся на данный момент в базе данных, и в зависимости от этого сохранить свои изменения поверх или отказаться от них. Естественно, это не требует от пользователя повторного заполнения всей формы (что очень раздражает пользователей).

## Трехуровневая архитектура

Приложение разделяется на три части: 1) управление интерфейсом пользователя; 2) выполнение правил обработки данных; 3) выполнение функций сохранения и выборки данных.

Данная архитектура позволяет четко разделить правила процессов и правила данных. Правила процессов реализуются исключительно на втором уровне. Этот уровень может представлять собой выделенный сервер приложений, который имеет право доступа к базе данных, а приложение пользователя обращается к базе данных только опосредовано. Одно из преимуществ такой архитектуры - использование нескольких СУБД в качестве хранилища данных. Все три части комплекса имеют фиксированные интерфейсы обмена данными, следовательно, имеет место изолированность уровня интерфейса пользователя от уровня базы данных за счет наличия ПО промежуточного слоя.

Современные СУБД позволяют реализовывать ПО промежуточного слоя как посредством мониторов транзакций (CICS, Enchina, Tuxedo и др.), так и на самом сервере баз данных в виде хранимых процедур и пакетов (частично или полностью). В этом случае код, реализующий интерфейс пользователя, не содержит вызовов предложений SQL, они «спрятаны» в код пакета или хранимой процедуры (здесь можно говорить об инкапсуляции, что некоторые авторы и делают). В таких случаях для решения пользовательского интерфейса применяют рекомендуемые, а не обязательные проверки правил. Данные при этом блокируются (они вообще не связаны с хранилищем данных) до тех пор, пока пользователь не захочет зафиксировать свои изменения (кнопка OK), а собственно изменения передаются атомарной транзакцией. Это позволяет эффективно использовать мониторы транзакций. Подобные решения рекомендованы для OLTP.

## Метрики генерации модулей

Одна из задач проектирования кода - оценить, сколько времени на это нужно и какие средства будут при этом использоваться.

В большинстве проектов оценку времени разработки производят дважды:

* на основе аналитической документации. Здесь привлекаются разработчики, но не на уровне: «За сколько сделаешь? - А за сколько надо?». Следует учесть не только средства разработки, но и аппаратное обеспечение. Это наиболее оптимистические оценки; хорошо, если они будут превышены только на 50%;
* после выполнения большей части проектирования схемы данных и модулей. Этот набор оценок предполагает более глубокую детализацию.

Если вы ни разу таких оценок не делали и вам непонятно, с чего начать, то разделите модули по группам: генерируется автоматически, простой, средней сложности, сложный, очень сложный.

Метрика - это таблица плановой трудоемкости (столько-то дней и столько-то человек требуется). В метрике учитываются как минимум три составляющие: проектирование модуля, генерация его кода, тестирование модуля (в которое входит и тестирование связей модулей). В метрику лучше включить больше условий - это станет своеобразной страховкой от отставания от графика. Следует учитывать как минимум следующие факторы:

* стабильность модели данных и степень ее изменения в течение разработки;
* стабильность модели функций и степень ее изменения в течение разработки;
* уровень квалификации персонала;
* пригодность выбранных средств разработки;
* использование автоматических генераторов кода (например, экранных форм и отчетов);
* соответствие среды требованиям средств разработки (станции разработчиков, серверы, сеть, операционные системы и т.п.).

Следует с очень большой осторожностью относиться к модулям, которые отмечены как сложные и для которых выбрано относительно низкофункциональное средство. Учтите также зависимости модулей и возможности накопления отставания от графика.

## Мегамодули

Это весьма распространенная особенность интерактивных систем. Создаются сложнейшие экранные формы с десятками страниц, один DML-запрос инициирует пару десятков триггеров и т.д. Задачу уменьшения сложности модулей сложно решить, если используются средства ускоренной разработки приложений. Решите, что вам нужно в этом модуле, а что - нет. И вряд ли мегамодуль - это то, о чем мечтал пользователь. Он вряд ли обрадуется, если будет листать страницы формы и уже на пятой забудет, что было в начале.

Мегамодули появляются и при пакетной обработке - они строят пакетные задания и отправляют их на обработку. Тестировать такие модули очень сложно. Возможно, что более простым решением окажется построение управляющей структуры для построения пакета.

Мегамодули в отчетах выглядят жутковато. Это происходит, когда проектировщики пытаются в одном отчете соединить несколько похожих отчетов. Пользователю совершенно все равно, каким количеством модулей обслуживается его отчет: чем меньше лишней информации он будет содержать, тем лучше.

В оперативных приложениях главная причина появления мегамодуля состоит в том, что аналитики указывают множество ненужных ограничений данных - например требования наличия в базе информации о покупателе только в том случае, если от его имени выполнен хотя бы один заказ товара. При обработке заказа это требование порождает проверку: «новый это покупатель или уже зарегистрированный в системе, и если новый, то когда его регистрировать». Можно разрешить покупателям «существовать» независимо от заказов, а если вам хочется, чтобы покупатели непременно были жестко связаны с заказами, - введите супертип «потенциальный клиент», и если такой клиент делает хотя бы один заказ, то он становится покупателем. Таким образом, вызовы двух экранных форм станут уже независимыми от реализации транзакций в базе данных.

## Макеты

При проектировании всегда возникает вопрос, стоит ли тратить драгоценное время на создание макетов. Это элементы (большие или маленькие) реализации реальных задач; они служат для демонстрации потенциальных функциональных возможностей, для изучения мнения пользователей. Макет - это средства представления идей в визуальной форме, поскольку, как говорят, лучше один раз увидеть.

Сколько функций выбрать для макетирования и каких, зависит от количества времени на выполнение этих работ и количества людей, которых можно к этому привлечь. Макет - это своего рода витрина для пользователей. Это означает, что большинство функций реализуют правила интерфейса. Успешная демонстрация может обеспечить заключение контракта на разработку, но есть и другая сторона вопроса: проследите, чтобы пользователь не принял внешнюю оболочку за готовую программу и не надеялся получить готовую систему через пару недель.

Другое назначение макетов - проверить проектные решения. Для этого годятся выявленные на этапе анализа критические участки системы. Хорошими вариантами будут несколько сложных отчетов; часть OLTP, часть OLAP. Это позволит привлечь к процессу проектирования группы тестеров, для того чтобы они проверили производительность системы.

## Описание

Для того чтобы разработчики могли правильно сгенерировать модуль, его техническая спецификация должна быть четкой и достаточно подробной.

Описание экранных форм и отчетов должно содержать:

* описание назначения формы (что делает);
* данные навигации (откуда вызвана, что может вызвать сама);
* формат вызова формы;
* список входных параметров и параметров по умолчанию;
* список выходных параметров и правила их обработки;
* описание обработки (события внутри модуля и их обработка);
* список ошибок, которые генерируются в процессе обработки формы и реакция на них;
* ограничения доступа к форме (каковы привилегии, разрешающие действия над формой и ее элементами, каковы привилегии, запрещающие эти действия).

Описание пакетных процессов должно содержать:

* описание функции, выполняемой пакетом;
* данные навигации (откуда вызван пакет, что может вызвать сам);
* формат вызова пакета;
* список входных параметров и параметров по умолчанию;
* список выходных параметров и правила их обработки;
* описание обработки (события внутри пакета и их обработка);
* список ошибок, которые генерируются в процессе обработки пакета и реакция на них;
* восстановление (обработка возникновения исключения в середине обработки пакета и реакция на него);
* вероятные блокировки (потенциальные конфликты и обработка ожидания);
* несовместимость с другими пакетными заданиями, отчетами и т.п. (эта информация для планировщика задач);
* регистрацию и аудит;
* ожидаемое состояние базы данных после выполнения пакета и проверку целостности данных;
* ограничения доступа (каковы привилегии, разрешающие вызов пакета, каковы привилегии, запрещающие этот вызов).

## Обработка ошибок

Обработка ошибок - это одна из подсистем, которая часто портит жизнь проектировщикам. Пользователи требуют вразумительных сообщений об ошибках. Им не понравится, если при попытке удалить поставщика информационная система выдаст сообщение вида «SQL0532N a parent row cannot be deleted because the relationship CLIENT\_ restricts deletion» вместо того, чтобы сообщить о невозможности удаления поставщика, если имеются факты поставки им товара на склад.

Рекомендуется сделать несколько уровней обработки ошибок. Первый уровень доступен в слое модулей, отвечающих непосредственно за вызовы SQL. Этот слой обрабатывает коды ошибок, передаваемые интерфейсом вызовов СУБД. Известно, что коды ошибок, детектирующие одно и то же нарушение ограничений данных, в разных СУБД разные. Поэтому, если ваше предложение имеет хотя бы вероятность работы с несколькими СУБД, вам придется интерпретировать коды возврата СУБД и построить матрицу соответствия кода возврата СУБД и кода ошибки, используемого в подсистеме обработки ошибок. Никакие модули, кроме интерпретатора кодов, не должны иметь доступа к кодам возврата СУБД. Вся обработка ошибок в информационной системе должна строиться на внутренних кодах возврата. Множество этих кодов может расширяться, но значения кодов изменять нельзя. Требуется проанализировать, в каком формате СУБД возвращает код. Это может быть:

* код ошибки, код ошибки операционной системы;
* sqlca.sqlcode, sqlca.sqlstate;
* sqlca.sqlcode, sqlca.sqlstate, код ошибки операционной системы.

Внутренняя ошибка может состоять из трех компонентов:

* sqlcode - собственно код ошибки;
* sqlstate - уточняющий код состояния и классификации ошибки;
* oscode - сопровождающий код ошибки операционной системы.

Первые два компонента обеспечивают поддержку стандарта, последний позволяет использовать эту же подсистему обработки ошибок для работы всех подсистем. Следует отметить, что некоторые СУБД возвращают код ошибки операционной системы в случае сбоев создания файлов, чтения страниц и т.п. Таким образом, подобная структура может обеспечить достаточную универсальность обработки ошибок. Тип ошибки - ошибка СУБД или ошибка определенного компонента информационной системы - будет контролироваться с помощью кода sqlstate. Как правило, sqlcode - это interger, sqlstate - это char(8) (длина выравнена на 4, что следует делать, например, для RS/6000, SUN SPARC, ALPHA, SGI), oscode - это integer. Например, код -532 интерпретируется в (-906, ’23001’, 0).

Второй слой обработки ошибок - это контекстная интерпретация внутреннего кода возврата. С каждым интерфейсным модулем может быть сопоставлен некоторый набор параметров интерпретации кода возврата. Например, для модуля редактирования списка поставщиков код (-906, ’23001’, 0) соответствует строке сообщения «нельзя удалить поставщика, поскольку есть связанные с ним поставки товара». По контексту этого исключения можно по требованию пользователя показать список поставок данного поставщика.