TIS/SQL подход к разработке информационных систем.

Создание высокозащищенных, объекто-ориентированных информационных систем на основе современных реляционных СУБД, с применением SQL и хранимых процедур.

(Манифест разработчика)

(и потенциальная тема для диссертации по специальности «05.13.19 Методы и системы защиты информации, информационная безопасность»)

Содержание

Задача манифеста

Этот манифест содержит обобщенное и универсализированное описание подхода к разработке *высокозащищенных*, *объекто-ориентированных* информационных систем, поддерживающих полное *ведение истории* всех изменений и воспроизводимость состояния данных на любой момент в прошлом. TIS/SQL подход базируется на архитектуре клиент-сервер и распространяется прежде всего на серверную составляющую этого тандема, именуемую «*ядро системы*». Для клиентской части даются только общие рекомендации по возможной архитектуре и процессу разработки.

Термин *TIS*, используемый в качестве родового имени для описываемого подхода, является аббревиатурой от *Trusted Informational System*, а полное наименование *TIS/SQL* – декларирует его приложение к разработке с использованием реляционных СУБД и языка SQL, одновременно подразумевая и возможность других приложений.

Назначение этой работы – в минимальном объеме изложить принципиальную схему системы, обладающей перечисленными качествами, а также правила ее проектирования и разработки, позволяющие в кратчайшие сроки достичь работоспособного прототипа. Этот документ является вводным для всех участников разработки и последующего сопровождения систем, создаваемых по технологии TIS/SQL. Первые две главы дают фундаментальное представление о конструкции системы и понятиях, положенных в ее основу, достаточное для оценки ее применимости к решению той или иной задачи. Последующие главы являются более детальной проработкой технических, организационных и управленческих вопросов, важных для реального и успешного воплощения подобной системы.

- Термин *высокозащищенный*, в контексте данного подхода, является синонимом англоязычного термина *trusted*, подразумевающего что система разграничения доступа для легитимных пользователей отвечает определенному набору требований, необходимых для хранения и обработки информации составляющей государственную, военную или коммерческую тайну.
- Термин *объекто-ориентированность* обозначает ориентированность на хранение и обработку документов сложной структуры, состоящих из множества строк в различных реляционных таблицах. Понятие *объект данных* является фундаментальным для системы разграничения доступа.
- *Воспроизводимость состояния данных* на любой момент в прошлом является следствием и подтверждением корректного решения задачи *ведения истории изменений*. С другой стороны, правильно реализованное *ведение истории*, выполняет также и функцию *аудита действий пользователей*, являющуюся обязательной для высокозащищенных информационных систем.

Все эти свойства системы полностью реализуются в виде базы данных (БД), именуемой «*ядро системы*» и находящейся под управлением сервера баз данных. *Ядро системы* полностью реализуется с применением таких средств современных реляционных СУБД как хранимые процедуры (stored procedures), функции (functions) и представления (views). Таким образом, непривилегированный пользователь и любые его программы могут непосредственно взаимодействовать с системой через интерфейс, предоставляемый СУБД для исполнения SQL запросов и вызовов хранимых процедур. Даже в том случае, если пользователи не будут допущены к БД напрямую, только реализация разграничения доступа и ведения истории в компактном ядре по строго определенным правилам обеспечивает их правильное функционирование, которое может быть протестировано и подтверждено за приемлемый срок.

Документ описывает промежуточный, но достаточно целостный результат концептуальной и практической проработки выше-перечисленных характеристик (высокозащищенность, объекто-ориентированность и ведение истории), а также вопросов их согласованного функционирования в виде фундамента информационной системы. Выбор конкретной СУБД сознательно оставлен за рамками этой работы, поскольку описываемый подход универсален и может быть адаптирован к любой СУБД, предоставляющей требуемую функциональность. Системы-предшественники использовали в своей основе MS SQL 6.x или Oracle 8.0.5, а в качестве основы для будущих систем предполагаются PostgreSQL (http://postgresql.org), FireBird (http://firebirdsql.org) и (с некоторыми оговорками) HSQLDB (http://hsqldb.org/).

1. Основные догматы

Первый догмат TIS – Модель данных, структура системы, система разграничения доступа и процесс разработки информационной системы должны быть спроектированы и согласованны между собой в единый комплекс до начала работы над ее предметной областью. Все дальнейшее проектирование должно осуществляться в терминах этого согласованного комплекса.

Следующие догматы представляют собой такой согласованный комплекс, являющейся частным решением поставленной задачи. При этом следует сразу отметить неравнозначность элементов входящих в этот комплекс – некоторые из них основополагающие, другие являются следствием первых, а третьи просто выбраны из множества допустимых, равнозначных решений и возведены в ранг догмы, чтобы пресечь споры о них на раннем этапе и избежать использования в разных частях системы разных решений для сходных задач. Также следует отдавать себе отчет в том, что некоторые декларируемые функции, и обеспечивающие их догмы, могут остаться нереализованными в силу своей невостребованности, однако разработка всегда должна осуществляться с учетом реализации этих возможностей, как только в них возникнет потребность.

Вообще говоря, под любую из приведенных догм, можно подвести более, или чаще – менее строгое основание, с привлечением международных и отраслевых стандартов, руководящих документов органов проводящих сертификацию, а также описаний реальных проблем, из личного опыта, на предотвращение которых направлено тот или иное требование. Однако такое обоснование существенно выходит за рамки выбранного формата изложения как по объему, так и по содержанию, и может быть интересно только небольшому количеству специалистов, да то – только после возникновения серьезного интереса к предложенному подходу.

1.1. Система разграничения доступа

Абстрагированная СРД – принципы функционирования системы разграничения доступа инвариантны к любой предметной области, для которой проектируется система. Правила разграничение доступа описываются на специализированном формальном языке, не зависящем от предметной области решаемой задачи, а их фактическая реализация основывается на специальных служебных полях, стандартным способом встроенных во все структуры для хранения пользовательских данных. Если язык оказывается недостаточен для решения какой-либо задачи, необходимые изменения вносятся в язык, и только как следствие – в реализацию.

Низкоуровневая реализация СРД – СРД реализуется на максимально низком слое системы, на котором возможно изолировать защищаемые объекты. Все последующие слои взаимодействуют с данными в интересах и от имени конкретного *субъекта доступа*, без необходимости заботится о разграничении доступа. Максимально возможное количество программного кода системы должно находится в условиях исключающих доступа к данным, минуя СРД.

Явное назначение прав доступа – любые права доступа, а также любые привилегии на специальный доступ к данным, должны назначаться любому *субъекту доступа* явным образом. Никаких априорных полномочий для пользователя-администратора системы не предусмотрено, т.е. понятие суперпользователя в системе отсутствует. Этот принцип позволяет избежать многих потенциальных уязвимостей и недостатков СРД.

Разграничение доступа на уровне типов объектов – Права доступа *субъекту доступа* могут быть назначены ко всем объектам одного типа в пределах некоторой, явно обозначенной, области, и «*областью данных*». При этом, субъект может иметь различные права доступа к различным объектам одного типа, находящимся в различных *областях данных* (т.е. - к объектам одинаковой структуры, данные которых хранятся в одних и тех же таблицах БД). Этот способ разграничения доступа наиболее употребителен для целевого класса информационных систем.

Разграничение доступа на уровне объектов – Права доступа могут быть назначены на каждый отдельный объект в системе. Такой способ разграничения наименее употребителен, однако его формальная осуществимость является необходимым требованием для соответствия определенным

классам защищенности. Кроме того, этот способ обеспечивает максимальную гибкость при разграничении доступа.

Разграничение доступа на уровне частей объектов – Субъект может иметь различные права доступа к различным частям (записям или полям) одного объекта.

Мандатное разграничение доступа – система должна обеспечивать разграничение доступа с использованием меток конфиденциальности информации, назначаемых объектам (документам), и уровней доступа к информации, назначаемых субъектам (пользователям).

Аудит безопасности – система, в процессе своего функционирования, должна сохранять достаточное количество информации для последующей независимой оценки событий, происходивших в ней, с точки зрения обеспечения информационной безопасности.

Применение концепции TCB. Trusted Computer Base (TCB) – совокупность программного и аппаратного обеспечения, являющаяся критичной с точки зрения защиты информации (как от несанкционированного доступа, так и от нарушения целостности). Части системы, находящиеся за пределами TCB, не могут получить больших полномочий, чем предоставлено действующему субъекту доступа, в интересах которого они выполняются. Т.е. для TCB ключевым является определение ее границ. СРД должна находится в окружении компактного ядра (TCB), в пределах которого ее корректное использование и функционирование является обязательным и поддающимся сертификации, а за пределами – неизбежным.

1.2. Объектная модель данных

Объекто-ориентированность – единицей хранения и изменения является *объект данных*, т.е. композитный документ состоящий из записей различной структуры. Для изменения *объекта данных*, используется парадигма доступа к файлу – открыть/изменить/закрыть, при этом изменения становятся доступными остальным пользователям только после «закрытия», а операция «закрыть» может сопровождаться с отказом от внесенных изменений.

Внутренняя числовая идентификация объектов – все *объекты данных* имеют уникальные и неизменяемые числовые идентификаторы, присваиваемые им при создании. Ссылки между *объектами данных* осуществляются с применением этих числовых идентификаторов.

Корректируемость любых прикладных данных – любые несистемные данные (в том числе ключевые), в любых документах, могут быть откорректированы. Наличие уникальных числовых идентификаторов *объектов данных* позволяет делать это не теряя уже установленных ссылок.

Версионность объектов. Система обеспечивает ведение истории всех изменений, фиксацию их времени и авторства, а также воспроизводимость любой версии любого объекта, и как следствие – всей БД на любой момент времени в прошлом. История изменений является одной из важнейших составных частей **аудита безопасности**.

Контроль качества версий объектов – контроль качества данных выполняется для каждой версии объекта, перед ее вступлением в силу. Изменения, не прошедшие контроль качества, могут быть продолжены или отменены, по выбору пользователя [их осуществляющего].

Блокировка объектов при редактировании – система обеспечивает блокировку объектов, запрещающую их изменение пользователем, отличным от наложившего блокировку. Блокировка не препятствует чтению последнего актуального состояния объекта, а также изменению других объектов, и может существовать сколь угодно долго. Блокировка может быть снята пользователем ее наложившим или пользователем имеющим специальную привилегию. Блокировка является естественной и неизбежной первой операцией при изменении (редактировании) объекта.

Символьное кодирование словарных данных. Для ограничения возможных значений некоторого поля конечным и перечислимым множеством, с ним ассоциируется именованный словарь, состоящий из списка кодов, являющихся допустимыми значениями, и их описаний – заменяющих или поясняющих эти коды при отображении документа. Все коды в системе являются символьными строками, по возможности состоящими из латинских букв верхнего регистра и цифр. При составлении словарей следует учитывать следующее множество универсальных кодов: { «Y» - Да/Истина; «N» - Нет/Ложь; «-» - нет значения/кода; «?» - значение неизвестно}. Для

использования числовых кодов – они преобразуются в строки. Использование символьных кодов позволяет без изменений использовать сложившуюся во многих предметных областей систему кодирования и делает данные соответствующих полей доступными для понимания даже без расшифровки. Символьная система кодирования снижает вероятность ошибочного использования неправильного словаря для какого-либо поля (за счет меньшего, чем у числовой системы кодирования, пересечения множеств значений разных словарей).

1.3. Модель базы данных и процесс разработки

Процедурное ядро является реализацией концепции TCB (Trusted Computer Base). Все операции по изменению данных в системе, и по управлению самой системой, реализуются в виде комплекса хранимых процедур, написанных на процедурном SQL (или другом языке) предоставляемом нижележащей СУБД, и осуществляющих в процессе своего выполнения контроль доступа и ведение истории изменений. Применение триггеров в данном подходе не предусмотрено и не рекомендуется, за исключением случаев, когда результат недостижим без них. Чтение данных обычно осуществляется через систему view, осуществляющих фильтрацию данных в соответствии с данными СРД. Комплекс процедур и система view составляют *ядро системы*. При необходимости, чтение данных может быть также реализовано через вызовы хранимых процедур. В качестве альтернативы встроенному языку процедурного SQL, или как дальнейшее развитие системы, ядро может быть реализовано как независимый от СУБД промежуточный слой-сервер.

Композитная модель БД подразумевает, что в пределах одной базы данных могут сосуществовать изолированные *области данных* одинаковой структуры, каждая из которых может рассматриваться как самостоятельная база данных. Понятие «*область данных*» имеет соответствующее отражение в СРД, которая учитывает принадлежность данных к конкретной области. При этом количество областей заранее не определено, все данные хранятся в одной структуре таблиц и могут рассматриваться и обрабатываться как единое целое. В качестве частного примера такой системы можно представить центральную базу данных компании, имеющей несколько складов, все операции на которых учитываются в центральной БД, при этом необходимо ограничить доступ персонала каждого склада только собственными данным. Одновременно, центральный менеджмент имеет доступ ко всей информации, как к единому целому.

Распределенность и репликативность в рамках композитной модели БД – Система должна разрабатываться с учетом возможности выделения некоторых *областей данных* в отдельные БД, с последующим обменом данными между ними, и периодическим слиянием их информации в единую центральную БД.

Автоматическая генерация шаблонного кода – любой шаблонный код должен порождаться только кодогенератором, на основе минимально необходимых исходных данных предоставленных на некотором формальном языке. Это прежде всего касается хранимых процедур ведения данных и view, т.е. - ядра системы, исходными данными для которого является структура *объектов данных* и реляционных таблиц, их составляющих. Данный принцип позволяет существенно сократить ручное кодирование и последующее сопровождение кода системы, избежать человеческих ошибок, сделать процесс внесения изменений в систему рутинным и прогнозируемым. Несоблюдение этого принципа ведет к существенному удорожанию разработки, а также к накоплению ошибок и расхождениям в алгоритмах решения одинаковых задач в разных частях системы, что затрудняет последующее сопровождение и развитие системы.

Использование системы контроля версий программного кода – разработка должна осуществляться с использованием системы контроля версий (CVS или аналогичной), которая должна обеспечивать воспроизводимость исходных текстов каждой значимой версии продукта.

Разделение процессов разработки и эксплуатации. Должно существовать минимум две инсталляции системы – одна для разработки и тестирования, а вторая – для эксплуатации. В эксплуатационную систему должны попадать только отлаженные и оттестированные версии продукта, снабженные инструкцией по переходу к данной версии от предыдущей.

2. Принципиальная схема системы

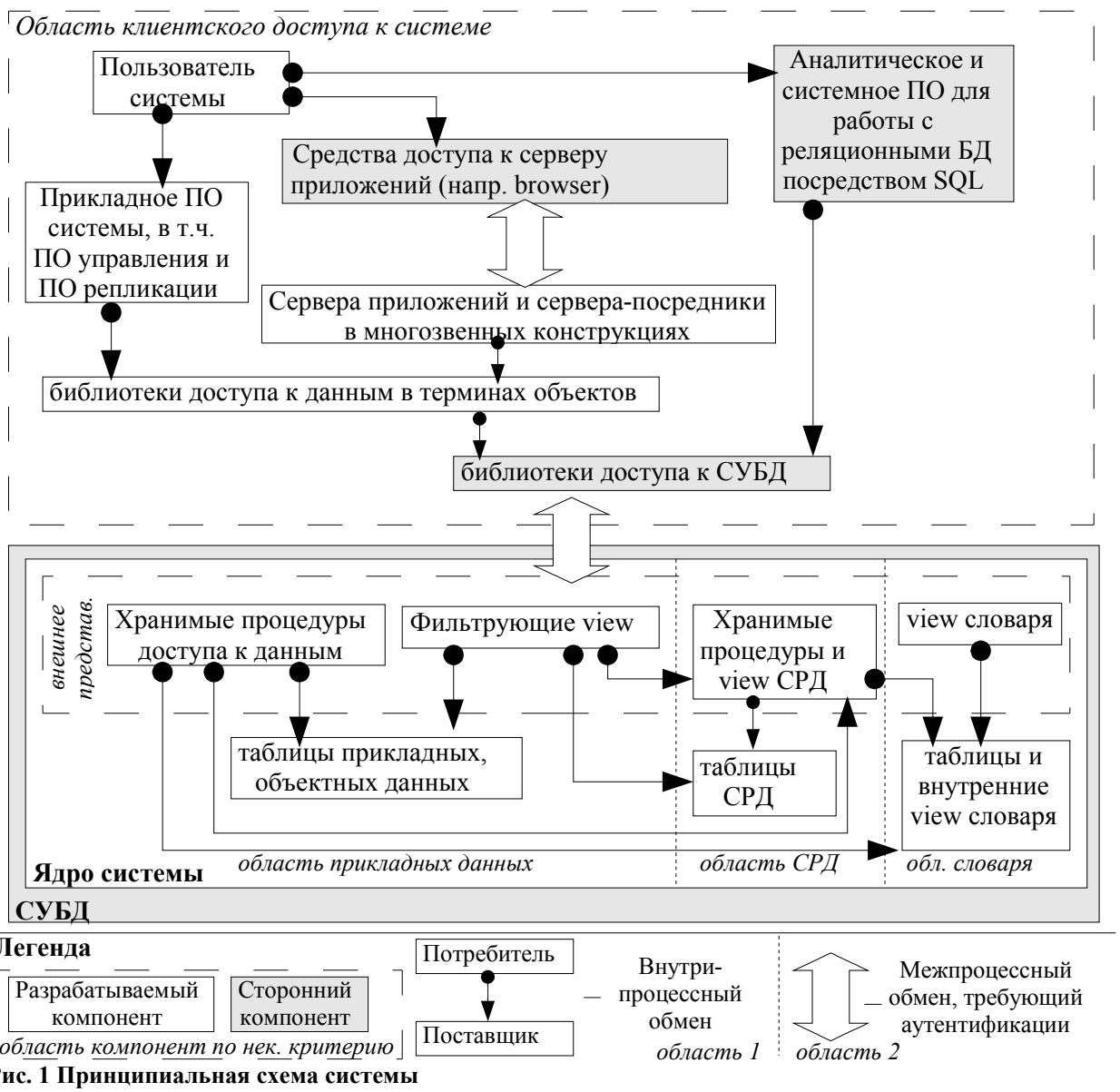


Рис. 1 Принципиальная схема системы

На рисунке представлена схема взаимодействия основных компонентов системы, а также их принадлежности к различным (перекрывающимся) областям внутри системы. Схема построена по иерархическому принципу, в виде направленного графа от основного потребителя информации, через последовательность промежуточных поставщиков, к самой информации в реляционных таблицах БД. (*Потребление* или *доступ к данным*, здесь и далее обозначает как доступ в режиме чтение, так и запросы на изменение.)

Под «*компонентом*» подразумевается некоторая совокупность исполняемого программного кода, занимающая строго определенное место в иерархии потребителей и поставщиков информации, и взаимодействующая с другими компонентами по строго определенным правилам, являющимся предметом отдельного согласования между разработчиками, результаты которого включаются в спецификацию на компонент-поставщик. Каждый компонент может разрабатываться отдельно, в режиме параллельной разработки, с использованием вместо необходимых компонентов «заглушек» или их стабильных, промежуточных версий.

Каждая область, охватывающая группу компонентов, подразумевает некоторый дополнительный набор правил и ограничений, преимущественно идеологического характера, которым должен подчи-

няться компонент, ей принадлежащий, при работе с прикладными данным и другими компонентами.

Два места взаимодействия компонентов на схеме обозначены особым образом – это т.н. «*межпроцессный обмен, требующий аутентификации*». В этих этих двух точках обеспечивается гарантированное соблюдение протоколов взаимодействия компонент, а следовательно – именно в этих точках осуществляется контроль и разграничение доступа к информации. Основным местом контроля и разграничения доступа, является взаимодействие клиентского ПО, через библиотеки доступа к СУБД, с объектами БД (процедурами и view), составляющими *ядро системы*.

Контроль доступа при взаимодействии с *серверами приложений* носит вспомогательный характер и позволяет повысить надежность и стабильность системы за счет изоляции пользователя от непосредственного взаимодействия с СУБД, по их пропертарным протоколам, которые, обычно, не заслуживают абсолютного доверия, или просто – не обеспечивают гарантированного соблюдения некоторых условий исполнения запросов. В вопросах разграничения доступа, сервера приложений должны преимущественно полагаться на механизмы, предоставляемые *ядром системы*, взаимодействуя с ним от имени *пользователя системы*, в интересах которого выполняется работа.

Во всех остальные точках взаимодействия компонентов, соблюдение согласованных протоколов и ограничений остается на совести разработчиков (а в области «клиентского доступа к системе» - на совести пользователей и администраторов) и может быть обеспечено только независимым контролем их работы. Таким контролем может являться сертификация ПО системы и аттестация объектов информатизации, его использующих. При сертификации, наиболее тщательному исследованию должен подвергаться код *ядра системы*, а также, при наличии серверов приложений – их код, и код используемых ими компонентов. Из выше сказанного вытекают особые требования к качеству и анализопригодности кода ядра системы (и серверов приложений), который должен быть по возможности линеен, прост, регулярен и хорошо документирован.

В ядре системы обозначены четыре основные области, первая из которых – «область внешнего представления» – объединяет хранимые процедуры и view, через которые (и только через которые) пользователи обращаются к данным в системе. Возможность пользователю обращаться только к этим объектам БД, и невозможность обращаться к реляционным таблицам напрямую, обеспечивается средствами СУБД. Все объекты этой области, являются частью одной из трех других областей.

Три оставшиеся области являются неперекрывающимися, а кроме того, для них соблюдается достаточно строгая однонаправленная ссылочность: *область СРД* ссылается на *область словаря*, а *область прикладных данных* ссылается и на область *СРД*, и на *область словаря*, но никогда наоборот. Все эти области можно разрабатывать параллельно, однако обычно разработка ведется в последовательности: словарь, СРД и область прикладных данных. Последняя, в реальных системах дробится на более мелкие тематические области-подсистемы, для которых уже и применяется параллельная разработка. Вкратце, назначения и характеристики этих областей следующие:

- *область словаря* – самая маленькая и архаичная, в простейшем случае состоит из одной таблицы, одного view и двух-трех процедур. Всего этого достаточно для хранения списков допустимых значений полей. Дальнейшее развитие словаря диктуется потребностями *прикладной области*.
- *область СРД* – достаточно небольшая; обычно включает около 10-ти таблиц; по одному view на таблицу; и 40-50 процедур. Этого достаточно для хранения списков пользователей, групп доступа и кодирования правил разграничения доступа. Выполняется в соответствии с классической реляционной моделью (т.е. является архаичной), не подразумевающей полного ведения истории. Последнее диктуется прежде всего практическими соображениями – таблицы СРД активно используются в запросах выполняемых через view прикладной области. Данная область является наиболее стабильной – единожды спроектированная, она очень мало изменяется в процессе развития системы, и с минимальными изменениями может быть перенесена в новый проект.
- *область прикладных данных* – самая большая и динамичная область, предназначена для хранения всех тех *объектов данных*, ради которых и разрабатывается система. В этой области в полной мере используется TIS подход (полное ведение истории, объектная организация данных, контроль доступа при любых действиях пользователя).

Что касается компонентов находящихся в области «клиентского доступа к система», то большая часть ошибок в их работе для разграничения доступа (и целостности данных) не критична, что позволяет при их разработке сосредоточится на решении прикладных задач, применяя при этом весь арсенал доступных средств, не заботясь о простоте кода, в ущерб его функциональности. Обычно, на эти компоненты приходится не менее половины всего кода системы, а их минимальное влияние на устойчивость системы и подконтрольность всех действий ядру – существенно упрощает и удешевляет процесс разработки, отладки и последующего сопровождения системы. В некоторых случаях, сертификацией этих компонентов можно полностью пренебречь, полагаясь на принцип: «Все разграничение доступа, а также контроль над допустимостью любых операций осуществляется *ядром системы*. Далее – все, что не запрещено – разрешено.»

2.1. Используемые технические приемы и особые средства СУБД

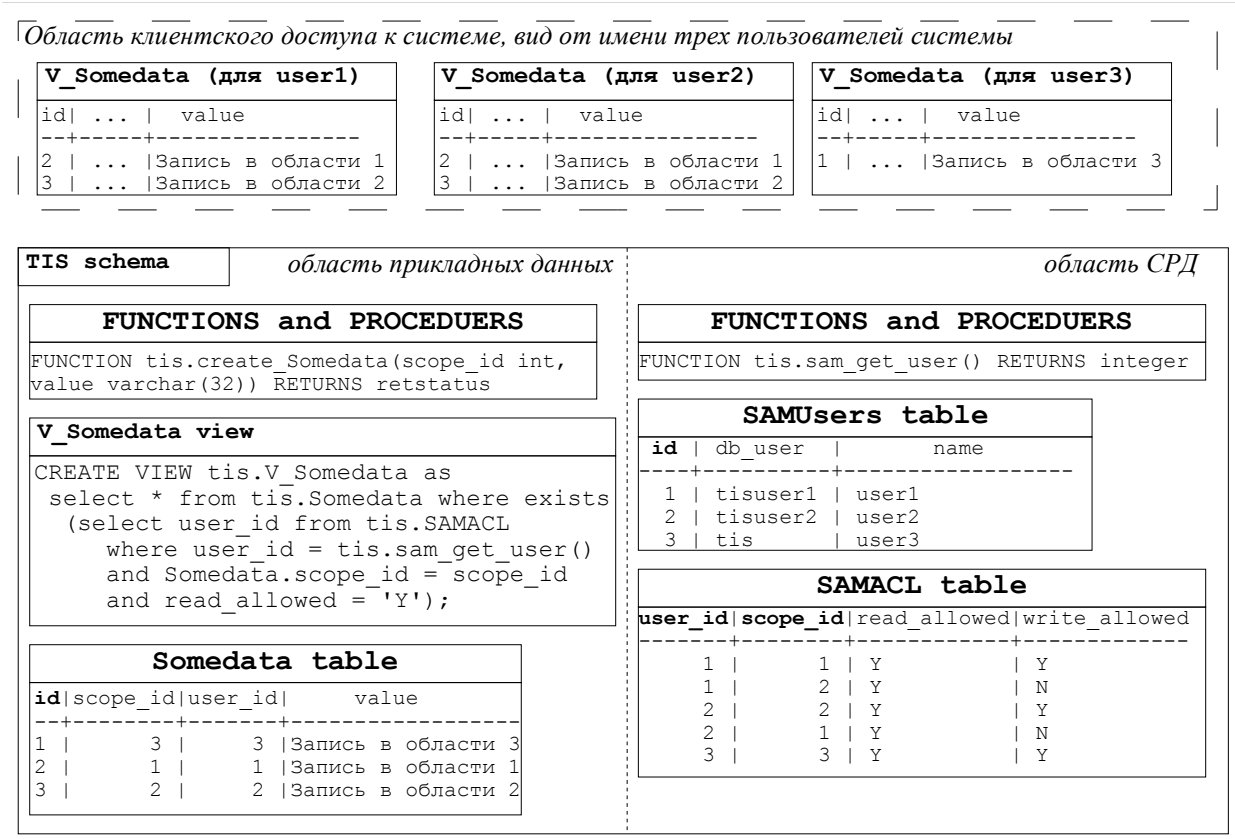


Рис. 2 Простейшая TIS

Лучший способ рассказать об используемых технических приемах и особых средствах СУБД, на которых основан TIS/SQL подход – реализовать простейшую TIS, и на ее примере показать что, где и каким образом применяется. С этого же начинается адаптация TIS подхода к любой новой СУБД.

На Рис. 2 представлена такая простейшая ИС, состоящая из 3-х таблиц, 2-х процедур и одного view. В «приложении В» приведен полный листинг ее реализации на PostgreSQL, и результаты исполнения тестовых операций по внесению и просмотру данных (Рис. 2), произведенные в диалоговом режиме, от имени трех различных пользователей БД, и демонстрирующие работу СРД.

Разработка любой TIS (да и вообще – любой ИС) должна начинаться с проектирования и реализации СРД, три кита которой – список легитимный пользователей, средство идентификации действующего пользователя и механизм назначения прав доступа.

Прежде всего необходимо реализовать список пользователей системы, независимый от списка пользователей БД, предусмотрев для каждого пользователя уникальный числовой идентификатор, используемый далее при ссылках на него, и при указании авторства действий. Кроме этого, необходима привязка пользователя системы к конкретному пользователю БД, равно как и возможность

изменить или ликвидировать эту привязку. (См. таблицу `SAMUsers` на Рис. 2)

Следующее – это функция `sam_get_user()`, идентифицирующая пользователя системы, в интересах которого выполняется текущий SQL запрос или процедура. Идентификация базируется на идентификации пользователя БД (`session_user`), которому принадлежит текущая сессия, и поиска в `SAMUsers` пользователя, ему соответствующего:

```
CREATE FUNCTION tis.sam_get_user() RETURNS integer STABLE
  AS 'select id from tis.SAMUsers where db_user = session_user'
  LANGUAGE SQL SECURITY DEFINER;
```

Тип данных поля `SAMUsers.db_user` зависит от того, какую информацию может предоставить конкретная СУБД. Предпочтительно использовать числовой идентификатор пользователя СУБД. Хотя он и не регламентируется стандартами на язык SQL, но доступен во многих СУБД и несет в себе меньше потенциальных проблем (например – с регистрозависимостью сравнения строк).

> **NB** В некоторых системах полезно реализовать возможность одному пользователю БД работать от имени различных пользователей системы, осуществляя переключение между ними. Это целесообразно, если вся работа осуществляется через промежуточный сервер приложений, всегда работающий с БД от имени одного и того же ее пользователя.

Последнее, что необходимо сделать, что-бы СРД приобрела завершенность – спроектировать и реализовать списки контроля доступа (ACL). В нашей простейшей TIS, права доступа назначаются непосредственно каждому пользователю, а критерием, по которому осуществляется разграничение доступа, является специальный признак принадлежности данных *области данных (scope)*, присутствующий в каждой записи в виде поля `scope_id`. Для каждой области предусмотрено два независимых права доступа – читать и писать. Здесь достаточно всего одной таблицы – SAMACL.

В реальной ИС следует как минимум: реализовать понятие *групп доступа*, которым приписываются ACL и которые, в свою очередь, назначаются пользователям; спроектировать такую структуру ACL, которая соответствует потребностям системы. (Базовая структура ACL TIS/SQL, которую можно взять за основу, будет рассмотрена позднее, в разделе посвященном SAM.)

Чтение данных осуществляется через систему view, в которые встроена фильтрация записей в соответствии с правами доступа действующего пользователя (см. `V_Somedata` на Рис. 2). Для фильтрации не только записей, но и полей внутри записей – соответствующие поля в операторе SELECT можно заменить на функции от этих полей и характеристик текущей записи, по которым функция определит права пользователя по отношению к данному полю, данной записи. Для полей, разрешенных к чтению, функция возвращает значение поля, иначе – NULL.

> **NB** Здесь следует сделать небольшое отступление, чтобы ответить на часто высказываемое сомнение в производительности фильтрующих view и предложение фильтровать данные где-нибудь за пределами СУБД. В действительности, фильтрация в SQL запросах (из которых и строятся view) – это самое высокопроизводительное из возможных решений, а передавать большие объемы данных за пределы СУБД – верный способ поставить крест на производительности любой системы, поскольку наиболее узким местом всегда является межпроцессный обмен, к тому же осуществляемый через сеть. И это не говоря уже о том, что фильтрация за пределами СУБД потребует и отказа от значительной части ее реляционной функциональности, самостоятельная реализация которой эквивалентна самостоятельной реализации СУБД. Тем не менее, полностью игнорировать снижение производительности запросов к фильтрующим view – нельзя, их скорость существенно зависит от оптимизатора SQL запросов, являющегося главным know-how любой реляционной СУБД. Сколь-либо достоверную оценку производительности можно сделать только на прототипе системы и только для конкретной композиции версии СУБД, операционной системы и аппаратного обеспечения.

Добавление, изменение и удаление записей, в TIS/SQL подходе, производится только через вызовы хранимых процедур или функций, исполняющихся с правами пользователя-владельца, допущенного к реальным таблицам и данным в них. Перед выполнением запрошенной операции, такая процедура осуществляет проверку прав доступа пользователя, ее вызвавшего, и допустимость переданных параметров с точки зрения целостности и непротиворечивости БД. Только после этого она вносит (или не вносит) запрошенные изменения в БД, фиксируя их время и авторство, а также предохраняя изменяемые данные. Предохранение измененных данных осуществляется в виде записей той же структуры, имеющих пометку о времени, начиная с которого запись утратила актуальность. В общем случае, хранение устаревших данных осуществляется в той же таблице, что и действующих, а фильтрующие view обеспечивают отображение записей актуальных на текущий (или запрошенный) момент времени.

В простейшей TIS, для иллюстрации процесса изменения данных, реализована процедура `create_Somedata()`, исходный код которой представлен в «приложении В». Чтобы не отвлекать внимания от главного, реализовано только добавление новых записей, которое не предусматривает ни хранения истории, ни контроля целостности и непротиворечивости – эти вопросы будут рассмотрены позднее.

Итак, суммируя вышеизложенное, главный технический прием – запрет прямого доступа пользователя к данным, и взаимодействие его с системой через view и хранимые процедуры, обладающие таким доступом. Особые средства СУБД – это view, хранимые процедуры и способность идентифицировать действующего пользователя БД. По результатом более детальной проработки, составлен список из 10-ти характеристик реляционных СУБД (см. Таб. 1), необходимых или желательных для применения TIS/SQL подхода. Адаптацию к новой СУБД следует начинать с анализа ее соответствия 6-ти обязательным требованиям из Таб. 1. (Отсутствие у СУБД обязательной функциональности – не является абсолютно фатальным, но адаптация TIS/SQL к подобным случаям – находится за рамками этой работы, и является предметом отдельного исследования конкретной СУБД и конкретного ТЗ.)

Таб. 1 Требования к функциональности СУБД

Функциональность СУБД	**использование**	**наличие**
1. Разграничения доступа на уровне пользователей БД	TIS-SQL подход базируется на том, что СУБД поддерживает список пользователей БД и позволяется разграничивать доступ к объектам БД (таблицам, view, процедурам и т.п.)	обязательно
2. Хранимые процедуры и функции, определяемые пользователем БД	Изменение (или чтение) данных в (из) реляционных таблицах, с предшествующей проверкой прав доступа. Ведение истории изменений и *журнала аудита*.	обязательно
3. view, зависящие от результатов идентификации пользователя БД	чтение данных из реляционных таблиц, с наложением фильтра, обеспечивающего выбор только тех строк, которые разрешены пользователю, исполняющему запрос	обязательно
4. Блокировка таблиц SQL запросами в хранимых процедур	Контроль качества данных, при их изменении через хранимые процедуры: контроль уникальности и непротиворечивости, требует механизмов синхронизации для процессов исполняющихся одновременно.	обязательно
5. Исполнение кода хранимых процедур и SQL запросов view с правами пользователя-создателя в интересах другого пользователя	Специализированный пользователь БД наделяется правами на чтение и изменение данных в реляционных таблицах. От имени этого пользователя создаются view и хранимые процедуры,составляющие *ядро системы*. Остальные пользователи должны получать доступ к данным только через *ядро системы*.	обязательно
6. Идентификация пользователя БД, выполняющего запрос ко view или хранимой процедуре	Каждая процедура или функция, перед выполнением запрошенной операции, идентифицирует пользователя БД, ее вызвавшего, определяет пользователя ИС, ему соответствующего, и проверяет наличие у того требуемых прав доступа.	обязательно

Функциональность СУБД	**использование**	**наличие**
7. Идентификация сессии с БД, в пределах которой происходит обращение ко view или хранимой процедуре	Возможность идентифицировать не только пользователя, но и сессию с БД, позволяет назначать одному и тому же пользователю различные права, в зависимости от характеристик сессии (например – IP адреса машины, с которой осуществлен заход). Кроме того, для систем в которых вся работа осуществляется через промежуточный сервер приложений, целесообразно прописать пользователем БД только этот сервер, и предоставить ему привилегию переключаться между пользователями системы в пределах одной сессии с БД, независимо от других, открытых им, сессий.	желательно
8. Возможность контролировать уровень изоляции транзакции в хранимых процедурах	Все приемы, используемые в рассматриваемом подходе, предполагают использование уровня изоляции READ COMMITTED, невозможность гарантировать его соблюдение непосредственно в хранимых процедурах ядра, потребует применения промежуточного сервера приложений.	желательно
9. Возможность контролировать факт завершения транзакции в хранимых процедурах	Ведение *журнала аудита* в таблицах БД, требует чтобы завершение транзакции, осуществленное в хранимой процедуре, не могло быть отменено после ее завершения. Последнее возможно, если СУБД поддерживает вложенные транзакции, и пользователь выполняет вызов к процедуре ядра внутри предварительно начатой транзакции, а потом отменяет эту "охватывающую" транзакцию. Все алгоритмы работы ядра системы, **следует согласовывать с пониманием этой проблемы.** Для журнала аудита, решением этой проблемы может стать использование внешних способов хранения данных, не зависящих от механизмов транзакций SQL.	желательно
10. Поддержка определения текущей даты и времени в UTC	Для каждой операции по изменению данных, фиксируется дата и время его вступления в силу, которое в дальнейшем может использоваться для изготовления срезов состояния БД на любой момент времени. Что-бы избежать проблем связанных с переходом на летнее/зимнее время, а также проблем с часовыми поясами в географически распределенных системах, может оказаться целесообразным использовать в них Универсальное Координированное Время (UTC)	желательно

2.2. Область применения

TIS/SQL подход, прежде всего, предназначается для интерактивных, документо-ориентированных систем, в которых пользователи взаимодействуют с единым централизованным хранилищем информации в режиме диалога, а все корректные изменения немедленно становятся общедоступными. Тем не менее, тот же подход может быть положен в основу при разработчике пакетных или комбинированных систем. (Так древнейшим предком TIS подхода была пакетная технология МИС, существовавшая в среде ЕС/ЭВМ, и обслуживавшая разнообразные задачи, от обработки больничных листов до вероятностного моделирования процессов распространения радиоактивного загрязнения.)

Не является догмой и единственность БД. Подразумевается, что несколько инсталляций системы могут эксплуатироваться независимо, при необходимости обмениваясь данными друг с другом. По мере изложения, в ключевых местах, обозначаются способы адаптации тех или иных понятий и конструкций к подобным «многоядерным» распределенным системам.

Очевидно, что область применимости того или иного инструмента начинается с тех задач, при работе над которыми он возник. Последние несколько проектов, на основе которых сформировался

TIS/SQL подход, были комплексными, интерактивными системами учета разнообразных ресурсов организации; хранения и обработки документов, порождаемых в процессе ее функционирования; а также – разнообразных связей между этими ресурсами и документами. По своему назначению и идеологии эти проекты близки к ERP системам.

Общим для этих систем являлось: сложность структур данных хранимых документов; развитая системы прямых и косвенных связей между ними; необходимость гибкой системы разграничения доступа, вплоть до разграничения доступа к индивидуальным документам и даже полям отдельных документов. Объем хранимой информации в БД – относительно небольшой, количество документов измерялось тысячами и десятками тысяч, а количество записей в отдельных реляционных таблицах – десятками и сотнями тысяч. Доступ в режиме чтения – преобладал над редактированием и добавлением, которые выполнялись преимущественно вручную.

Одной из проблем, оказавшей существенное влияние на TIS подход, явилась необходимость сертификации одной из систем-предшественников (KI-MACS) в Гостехкомиссии при Президенте РФ, по требованиям для систем обработки информации, составляющей государственную тайну. Именно в процессе осмысления этой задачи, а также недостатков выполненных решений, сформировались предложенные здесь принципы разграничения доступа и подходы к их реализации средствами языка SQL. Одновременно, идеология ведения истории изменений были унаследованы от систем предшественниц, разработанных предыдущими поколениями программистов.

Таким образом, первая и наиболее очевидная область применения – это ERP-подобные системы, а также системы по обработке конфиденциальной информации, в т.ч. составляющей гос-тайну, требующие сертификации на соответствие формальным требованиям по разграничению доступа и защите информации.

Помимо «тяжеловесных» систем, TIS/SQL подход может быть использован практически для любых документо-ориентированных бизнес-систем, изначально наделив их характеристиками, позволяющими развиться в систему масштаба предприятия.

Отдельной нишей могут стать системы абонентского обслуживания, биллинга, электронной коммерции, Internet сообщества с индивидуальными профилями пользователей, а также системы для совместной публичной работы по формированию общественных структурированных информационных массивов (подобных Wikipedia.org). Так понятие *области данных*, как отдельно защищаемой совокупности некоторых документов, может использоваться для реализации в БД профилей пользователей, необходимых в любой из этих систем, а изначальная реализация СРД, учета изменений и их авторства в компактном ядре – позволит обойтись без большинства неустранимых архитектурных проблем, которыми страдают многие подобные системы.

2.3. Основные понятия

Большой, и зачастую неочевидной проблемой, возникающей при освоении новой идеологии, является различное понимание одних и тех же терминов, разными участниками команды. Так любая отрасль или идеология, беря на вооружение некоторое понятие и слово, его обозначающее, наделяет этот тандем новыми характеристиками и значениями, одновременно отказываясь от, или уменьшая вес, значений и характеристик, ранее им присущих. Таким образом, профессиональный язык переполнен омонимами, споры о которых способны пожирать рабочее время часами, днями и неделями, особенно в первые месяцы работы над новым проектом. Минимизация подобных споров на начальном этапе, способна существенно сократить время на разработку прототипа.

Далее приводятся определения и описания всех ключевых понятий, составляющих TIS/SQL, включающие их происхождение, назначение и взаимосвязи друг с другом. Собственно, совокупность этих описаний и составляет TIS подход, в то время как последующие главы этой работы – посвящены деталям его реализации с применением SQL, и рекомендациям по планированию и ведению проекта. Две концепции являются фундаментом TIS подхода – это *объект данных (ОД)* и *область данных*. Все остальные понятия, в той или иной мере, базируются на двух этих «китах». Третий «кит» это *система разграничения доступа* (СРД или SAM), являющаяся согласованным набором принципов, идей и структур данных, превращающих обычную информационную систему (IS), в высокозащищенную (или – трастовую) ИС (т.е. TIS).

Для большинства понятий первым дается англоязычное или аббревиатурно-идиоматическое название, под которым оно будет фигурировать в исходных текстах, а следом за ним – русскоязычный эквивалент.

NB Все понятия формулируются и рассматриваются с некоторым уклоном в область реляционности и SQL-ориентированности, обусловленным темой работы. Тем не менее, их применимость не ограничивается построением ИС только на основе SQL. Все эти идеи, и прежде всего – «три кита», представляют самостоятельную ценность, и составляют целостную систему понятий, достаточную для построения высокозащищенной, объекто-ориентированной ИС на любой основе.

NB При дальнейшем чтении этого раздела следует быть особенно внимательным к тем понятиям, которые кажутся известными по предыдущему профессиональному опыту, поскольку они никогда не являются в полной мере тем, чем кажутся поначалу.

2.3.1. Data object - объект данных (ОД)

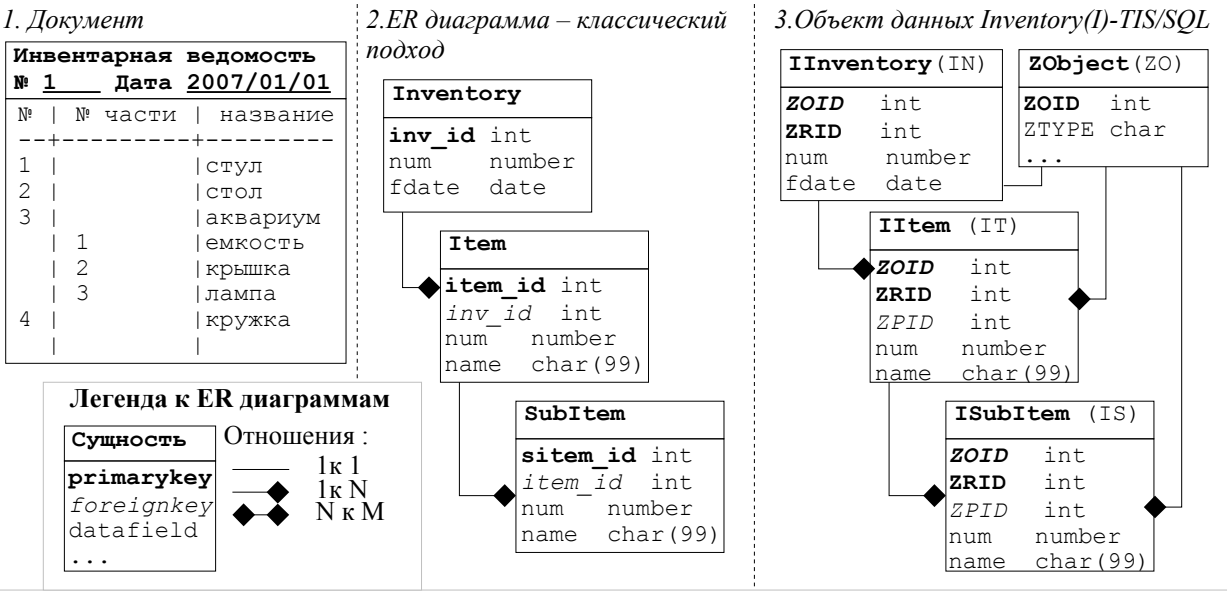


Рис. 3 Понятие объекта данных. Эволюция

Объект данных (Data object, ОД) – это целостный документ, имеющий четкие границы и правила обращения. В TIS/SQL этому соответствует совокупность записей в различных таблицах БД, составляющая единый документ. Все эти записи имеют общий идентификатор ОД (Object ID). Между собой записи ОД могут быть связаны в различные конструкции, наиболее распространенной из которых является – жесткая иерархическая система отношений `parent/child`, с одной явной заголовочной записью объекта, которой прямо или косвенно подчинены все остальные.

Изменение актуального состояния *объекта данных* всегда происходит синхронно, и всегда сопровождается проверкой его допустимости – *контролем качества* (*Quality Control*, *QC*). Над любым *ОД* определены следующие базовые операции, определяющие алгоритм его изменения:

1. *Создать* (`create_object`) – создает новый ОД и переводит его в состояние «открыт для изменений» (подробнее см. операцию *открыть*).
2. *Открыть* (`open_object`) – переводит существующий ОД в состояние «открыт для изменений». Дальнейшее изменение доступно только пользователю, выполнившему эту операцию, а все внесенные изменения становятся доступны остальным пользователям только после успешного выполнения операции «Закрыть». Одновременно ОД может быть открыть только одним пользователем.
3. *Закрыть* (`close_object`) – переводит объект в нормальное состояние, после которого он может быть снова открыт. Закрытие может происходить, по выбору пользователя, по двум сценариям: в

первом, новое состояние ОД успешно проходит процедуру контроля качества, и замещает предшествующее; во втором – все внесенные изменения отменяются.

4. *Удалить* (`delete_object`) – ОД (все его записи) помечаются как утратившие актуальность с текущего момента времени. Операция удаления – атомарная, т.е. не требует «открытия» объекта.

Для каждого *типа записей* (каждой таблицы), должны быть реализованы следующие операции, осуществимые только над *открытым* состоянием ОД: *создать запись* (`create_<record>`), *изменить запись* (`update_<record>`), *удалить запись* (`delete_<record>`). Удаление записи подразумевает рекурсивное удаление всех подчиненных ей запией: так удаление записи `IItem` (Рис. 3), ведет к удалению всех подчиненных ей записей `ISubItem`.

Теперь, сформулировав понятие *объекта данных*, следует вернуться к Рис. 3 и рассмотреть с его помощью эволюцию понятия и его реализаций, поскольку без знания проблем и ошибок, под влиянием которых она происходила, есть риск их повторить.

Прежде всего, обратимся к исходному, «бумажному» документу, – гипотетической «Инвентарной ведомости», имеющей три уровня подчиненности структур: «шапка» документа, учетная единица и составные части учетной единицы. Существуют СУБД, в которых такой сложный документ может быть помещен в одну запись. Например, технология МИС (Мета-ядро Информационных Систем, см Приложение А) базировалась на СУБД ADABAS™, которая это позволяла, и осознанной потребности в специальном понятии *объект данных* не возникало.

При переходе к реляционным СУБД, документ должен быть подвергнут декомпозиции, а его содержимое – разложено по трем таблицам, представленным на ER-диаграмме классического подхода (все таблицы сразу снабжены суррогатными ключами, позволяющими безболезненно корректировать любые прикладные данные, без потерь ссылочной целостности, что, вообще говоря, является отдельным эволюционным шагом). Такое преобразование дает новое качество – части, ранее принадлежавшие разным документам, можно легко комбинировать, анализировать и собирать в новые документы, любой структуры. Однако, при этом утрачена целостность документов, с которыми должны работать пользователи. Обеспечить всю комбинаторику возможных форм документов полноценным пользовательским интерфейсом просмотра и редактирования – невозможно, а отдельные записи – слишком атомарны и бессмысленны в отрыве от своего окружения.

Таким образом, возникает потребность определить базовые формы документов, пригодные для использования в прикладном ПО, которое при загрузке такого документа из БД, будет осуществлять сборку записей из различных реляционных таблиц, а при сохранении изменений – осуществлять обратный процесс. Именно для таких базовых форм документов возникло понятие *объект данных*, а вместе с ним – целый букет проблем:

1. Как убедиться, что между выгрузкой документа из БД и сохранением измененного состояния, никакая его часть не подверглась изменению? В действительности, единственное надежное решение – заблокировать все таблицы (именно таблицы – поскольку добавление записей, это тоже изменение), в которых могут находится записи входящие в документ, еще раз его выгрузить и сравнить с тем, что было выгружено в прошлый раз (а этого мы, кстати, обычно уже не имеем, поскольку обладаем только измененным состоянием документа).
2. Как предупредить других пользователей, что все записи, входящие в этот документ не должны подвергаться изменению, поскольку он уже редактируется некоторым пользователем? При этом, поскольку документ может редактироваться часами – пользователи должны иметь возможность читать любые данные из БД и изменять любые другие документы.
3. Как обеспечить согласованное состояние БД в любой момент, если оно возможно только тогда, когда каждый документ удовлетворяет некоторым критериям качества, проверка которых осуществима только по совокупности всех записей его составляющих. При этом, операции над каждой записью являются отдельным вызовами хранимых процедуры, и вообще говоря – должны (или по крайней мере – могут) выполняться – каждая в рамках отдельной транзакции.
4. Как обеспечить равную стоимость запросов к различным записям такого составного документа. Например, при загрузке инвентарной ведомости по ее `inv_id`, для таблиц `Inventory` и `Item` в запросе будет участвовать по одной таблице – обе они содержат поле

`inv_id`, одна в качестве первичного, другая в качестве вторичного ключа. А вот при обращении к таблице `SubItem`, уже потребуется использовать две таблицы, поскольку определить принадлежность ее записей конкретным ведомостям можно только с привлечением таблицы `Item`. Эта проблема становится особо ощутима при написании фильтрующих view, если критерием разграничения доступа является принадлежность записи конкретному документу, что плавно перетекает в проблему следующую:

5. Как обеспечить унификацию алгоритмов обработки разных типов записей, в разных таблицах, и как не запутаться, какие записи каким записям подчиняются, и какие из них «равнее других», поскольку являются «шапками» документов, обозначая не только себя самое, но и весь документ целиком, и в каких случаях...

Решением этих проблем, выработанным за несколько итераций, является приведенное ранее определение *объекта данных* (и правил обращения с ним), иллюстрируемое третьей частью Рис. 3.

Во-первых – введена специальная служебная таблица `ZObject`, в которой учитываются все ОД и их состояние, вне зависимости от того, в каких таблицах хранятся составляющие их записи. Теперь любую операцию над объектом можно начинать с проверки записи о его состоянии в таблице `ZObject`, а процесс синхронизации и блокировки становится тривиальным и универсальным. Такими же универсальными становятся рамочные процедуры для базовых операций *создать/открыть/закрыть/удалить*.

Во-вторых – все служебные поля во всех таблицах унифицированы и сгруппированы в начале записи, а идентификатор объекта данных (ZOID) распространен по всем записям ОД. Таким образом, все они находятся в «одном шаге» от заголовочной записи `ZObject` и друг от друга. Эта группа стандартных полей получила название *стандартный заголовок* (`STD_HEADER`), и включает набор полей, предназначенных для организации жестких отношений parent/child:

- ZRID (Row ID) – уникальный идентификатор этой записи, среди других записей той же структуры в этом объекте данных. Пара полей {ZOID,ZRID} – однозначно идентифицирует запись и образует первичный ключ (пока мы не рассматриваем хранение истории изменений, которое потребует расширения первичного ключа)
- ZPID (Parent ID) – идентификатор ZRID той записи этого же ОД, которой подчиняется данная запись в рамках жестких отношений parent/child. {ZOID,ZPID} однозначно идентифицирует отцовскую запись, а в какой таблице она находится – предопределено структурой отношений внутри *объекта данных*.

NB Следует обратить внимание, что система отношений parent/child хоть и является доминирующей, при проектировании структуры документов, но вовсе не обязательной. Например, для документа на Рис. 3, можно отказаться от отношений подчиненности `IInventory/IItem`, которая становится очевидной при единственности записи `IInventory`, после чего, обратив внимание на идентичность структуры таблиц `IItem` и `ISubItem` – можно отказаться от последней, организовав иерархические отношения между самими элементами `IItem`. При этом мы получаем возможность сколь угодно глубокой детализации по составным частям, а алгоритмы загрузки/сохранения/удаления такого документа – не претерпят принципиальных изменений, поскольку, для обращения ко всем записям объекта в таблице, достаточно одного запроса с параметром ZOID на таблицу. Этот пример хорошо демонстрирует разделение между универсальными структурами данных/алгоритмами для хранения/извлечения/разграничения доступа, и связями внутри документа, специфические алгоритмы обработки которых можно разрабатывать отдельно, например – вообще за пределами СУБД, используя для этого более подходящие средства разработки прикладного ПО.

Решение проблемы согласованности и контроля качества по совокупности объединено с решением задачи хранения истории, о котором будет рассказано отдельно. Основная идея состоит в том, что в одной таблице хранятся несколько версий каждой записи: неограниченное количество относящихся к прошлому; одна (или ни одной) – к настоящему; одна (или ни одной) – к будущему.

Фильтрующие view отображают ту версию, что является действительной в текущий (или в запрошенный) момент времени. Таким образом, все изменения в «открытом» объекте данных относятся к «будущему совершенному», но станут «настоящим» только тогда, когда пользователь этого потребует, и только если пройдут *контроль качества* по совокупности.

Следующий важный эволюционный момент – это система кодов для типов ОД и таблиц, а также правила именования таблиц, входящих в структуру *типа объекта данных*. В таблице `ZObject` на Рис. 3 имеется поле `ZTYPE`, которое используется для указания – какой именно тип ОД стоит за этой записью. Следовательно — каждый тип должен иметь уникальный код. Имя типа обычно происходит от имени заголовочной таблицы, в данном случае – `Inventory`. При небольшом количестве типов длину кода можно ограничить одним символом, а выбор начинать с первой буквы названия типа (если она, конечно, свободна). В данном случае *код типа объекта данных* – `"I"`. Далее, этот код используется в качестве аффикса, для обозначения принадлежности других именованных структур к структуре данного *типа объекта данных.* Так все таблицы получают *код типа объекта данных* в виде префикса к своему названию и в качестве первого символа *кода таблицы*. Таким образом устраняется путаница между названием таблицы и названием *типа ОД*: теперь `Inventory` – это *тип объекта данных*, а `IInventory` – одна из таблиц входящая в его структуру. Коды таблиц рекомендуется использовать в качестве имен переменных, для соответствующих структур данных, и в качестве синонимов, при написании SQL запросов, что существенно повышает читаемость их условий и повторное использование программного кода. Так, например SQL запрос вида:

```
SELECT "IN".ZOID, "IN".num from IInventory "IN", IItem IT, ISubItem "IS" where
"IN".ZOID=IT.ZOID and IT.ZOID = "IS".ZOID and IT.ZRID="IS".ZPID and
"IS".name='лампа' and IT.name<>'аквариум'
```

гораздо легче поддается анализу, чем при использование в качестве синонимов `T,T2,T3` и т.д.

Специальный код "Z" используется для обозначения служебных полей и системных структур данных (например – таблица ZObject, поля ZOID,ZRID,ZPID). Другой специальный код "X" – предназначен для структур СРД, и третий, "S", для «специальных», «необъектных» данных. Код "Y" – зарезервирован для введение кодов *типов ОД* переменной длины, если это потребуется.

Резюмируя все выше сказанное, следует расставить терминологические акценты для всех понятий, связанных с *объектом данных*:

- *объект данных* (ОД) – совокупность записей, составляющая единый документ конкретного типа.
- *тип объекта данных* – именованная категория документов, хранимых и обрабатываемых ИС.
- *структура таблиц ОД* – совокупность таблиц БД, для хранения всех ОД, определенного типа.
- *имя типа объекта данных* – уникальное символьное осмысленное обозначение *типа ОД,*.
- *код типа ОД* – уникальный, короткий (ограниченной длины) синоним для *имени типа ОД*.

На практике, особенно в разговорной речи, вместо всех этих пяти терминов, часто используется слово «*объект»*, что бывает достаточно для взаимопонимания, но некорректно по форме. Во всех документах следует придерживаться вышеприведенной терминологии, а термина «*объект*», в значении ОД, – избегать.

> **NB** В заключение, следует расставить все точки над i, в отношениях между понятием *объект данных (ОД)* и объектно-ориентированным программированием. В этой работе термин «объекто-ориентированный» сознательно противопоставляется термину «объектно-ориентированный», а термин «тип ОД» — термину «класс». Реализация понятия ОД не требует со стороны СУБД никаких объектно-ориентированных расширений. Понятие ОД не подразумевает наследования (inheritance), которое многими разработчиками считается неотъемлемой тенью слова *объект*. Вместо наследования, возможна реализация подтипов документов на уровне *контроля качества* по совокупности данных, обеспечивающая хранение разных типов документов в одной и структуре таблиц ОД. Для реализации понятия ОД в прикладном ПО рекомендуется использование объектно-ориентированных свойств выбранного языка программирования, однако наличие таких свойств – не является строго необходимыми.

2.3.2. Object version – версия объекта

№	Поле	Тип	Описание
01	**ZOID**	bigint	уникальный идентификатор ОД
02	**ZVER**	bigint	уникальный номер версии ОД
03	ZTYPE	char(1)	код типа ОД
04	ZSID	bigint	идентификатор *области данных*, к которой принадлежит ОД
05	ZLVL	smallint	уровень секретности ОД (используется мандатной СРД)
06	ZUID	bigint	идентификатор пользователя, создавшего эту версию ОД
07	ZSTA	char(1)	статус версии N – действующая, C – устаревшая, D – ОД удален и т.д.
08	ZDATE	timestamp	дата/время вступления в действие данной версии ОД
09	ZDATO	timestamp	дата/время прекращения действия данной версии ОД

Таб. 2 ZObject (ZO) – учет версий ОД

Принцип версионности ОД: Любое изменение ОД возможно только с порождением новой его версии. Таблица `ZObject` (Таб. 2), введенная для хранения служебной информации обо всех ОД, содержит по одной записи на каждую версию. Ее первичным ключом является комбинация полей `{ZOID,ZVER}`. Поле `ZSTA` (Таб. 3) определяет текущий статус версии, а поля `[ZDATE, ZDATO)` указывают диапазон времени, в течении которого версия была действительна (т.е. имела `ZSTA='N'`).

Запись `ZSTA='N'` – имеет `ZDATO=NULL`, т.е. дата завершения для этой версии еще не известна. Перекрытия диапазонов `[ZDATE, ZDATO)` между версиями `ZSTA∈{N,C,D,U,M}` – не допускается. Версия `ZSTA∈{I,L,T,P}` – является временной, блокирующей появление любых других версий ОД. Временные версии могут переходить в постоянные или полностью удаляться. Значения `{R,A,O,W,F}` зарезервированы как альтернатива удалению временных версий, и для них допускается перекрытие `[ZDATE, ZDATO)` с последней из предшествующих версий `{N,C,M}`.

Таб. 3 Допустимые значения поля ZSTA

код	значение	пере-ходит в	Описание
N	New	C,M	Действующая версия ОД (только одна версия с таким статусом)
C	Corrected		Устаревшая версия
R	*Rollback*		*Отмененная транзакция (зарезервированный код)*
P	*Prepared*	*N,R*	*Прошло QC и готово к N (зарезервировано для распред. транзакций)*
D	Deleted	U	ОД удален (только одна версия с таким статусом)
U	Undeleted		Удаленный ОД восстановлен
I	Intermediate	N,R	ОД заблокирован для редактирования
L	Locked	F	ОД заблокирован без редактирования
F	*Free*		*Блокировка ОД снята (зарезервированный код)*
T	Transit	A,O,W,M	ОД предназначен для передачи, его редактирование – запрещено
A	*Accepted*		*Успешно принято из транзита (зарезервированный код)*
O	*Offcast*		*Отвергнуто принимающей стороной (зарезервированный код)*
W	*Withdrawn*		*Отозвано из транзита (зарезервированный код)*
M	Moved		ОД перемещен в другую область данных, или у ОД изменен ZLVL

Любая запись ОД, также может иметь несколько версий, и каждая версия записи действительна для некоторого диапазона версий ОД, который указывается в специальных полях стандартного заголовка (STD_HEADER), рассматриваемого в следующем разделе.

Алгоритм внесения изменений в ОД таков:

1 Имеется действующая версия ZVER=X:{ZSTA='N', ZDATO=NULL}

2 Открываем ОД: создается новая версия ZVER=(X+1):{ZSTA='I', ZDATE=NOW, ZDATO=NULL}

3 В записи ОД вносятся изменения, вступающие в действия начиная с версии (X+1)

4 Закрываем ОД: Запускается процедура завершения изменений

4.1 Выполняется *контроль качества* нового состояния ОД, и если он успешен, то производиться:

4.2 ZVER=X:{ZSTA='N', ZDATO=NULL} ==> {ZSTA='C', ZDATO=NOW}
ZVER=(X+1):{ZSTA='I', ZDATE=NOW, ZDATO=NULL}==>{ZSTA='N', ZDATE=NOW, ZDATO=NULL}

4.3 Иначе, если контроль качества неуспешен – следует вернуться к изменению ОД или отказаться от уже внесенных изменений, ликвидировав временную версию (X+1), и все версии записей к ней относящиеся.

Блокировка ОД – операция родственная *открытию для изменений*, но вместо ZSTA='I', используется ZSTA='L', никакие изменения не возможны, а единственный вариант завершения – удалений этой временной версии.

Удаление ОД является атомарной операцией, в процессе которой (помимо логического «удаления» записей) выполняется:
ZVER=X:{ZSTA='N', ZDATO=NULL}==>{ZSTA='C', ZDATO=NOW}
ZVER=(X+1):{ZSTA='D', ZDATE=NOW, ZDATO=NOW}

Что касается *транзита* и *перемещения*, то здесь возможны существенно различные реализации, зависящие от семантики того реального явления, которое должно стоять за этими абстракциями. Само понятие *транзит* рассматривается далее, в соответствующем разделе.

2.3.3. STD_HEADER – стандартный заголовок

№	Поле	Тип	Описание
01	**ZOID**	bigint	уникальный идентификатор ОД
02	**ZRID**	bigint	уникальный, в пределах ОД, идентификатор записи этого типа
03	**ZVER**	bigint	номер версии ОД, начиная с которого вступила в действие версия записи
04	ZTOV	bigint	номер версии ОД, начиная с которого прекратила действие версия записи
05	ZSID	bigint	идентификатор области данных, к которой принадлежит ОД
06	ZLVL	smallint	уровень секретности ОД (используется мандатной СРД)
07	ZPID	bigint	идентификатор ZRID записи parent, принадлежащей этому же ОД

Таб. 4 STD_HEADER – Стандартный заголовок

Стандартный заголовок – это группа полей, которая присутствует во всех таблицах БД, предназначенных для хранения прикладных (несистемных) данных. Стандартный заголовок предшествует всем остальным полям записи и предназначен для унификации алгоритмов добавления/изменения/удаления записей, включающих ведение истории изменений. Он содержит всю необходимую информацию о положении данной записи в ОД, ее актуальности, а также все метки и характеристики, необходимые для разграничения доступа к этой конкретной записи.

Наиболее употребительный состав полей стандартного заголовка для записи входящей в ОД представлен в Таб. 4 (т.н. тип STD_HEADER). Происхождение и назначение полей ZOID,ZRID,ZPID – подробно изложено в разделе 2.3.1. Поля ZSID,ZLVL – всегда дублируются из записи ZObject, и предназначены для ускорения работы СРД. А поля [ZVER,ZTOV) – задают диапазон версий ОД, для которого действительна данная версия записи. Поля {ZOID,ZRID,ZVER} – составляют первичный ключ таблицы, снабженной стандартным заголовком этого типа. При необходимости, стандартный заголовок можно дополнить и другими полями, например полями ZDATE и ZDATO, которые позволят упростить и ускорить построение срезов состояния БД на произвольный момент времени.

Особое внимание следует уделить возможным значениям поля ZTOV:

- ZTOV>0 – запись устарела начиная с версии ZTOV
- ZTOV=0 – запись действительна в настоящий момент
- ZTOV<0 – запись относится к незавершенной логической транзакции, представленной в таблице ZObject записью версии с ZSTA='I'. Возможны следующие значения для незавершенных записей:
  - `ZTOV=-1 (ZC_ZTOV_UPDATED)` измененная версия записи (должна заменить предыдущую);
  - `ZTOV=-2 (ZC_ZTOV_DELETED)` запись удалена;
  - `ZTOV=-3 (ZC_ZTOV_NOCHANGES)` запись без изменений;
  - `ZTOV=-4 (ZC_ZTOV_RDELETED)` запись удалена, вследствие удаления записи parent.

На основе полей стандартного заголовка, фильтрующие view отображают только те записи, которые разрешены к просмотру действующему пользователю, и являются действующими в указанный момент времени (в простейшем случае – действ. в настоящий момент, т.е. имеющие ZTOV=0).

Для некоторых видов данных, полная объектная структура, требующая для простейшего докумен-

та минимум две записи в двух различных таблицах, и еще одну дополнительную запись в ZObject при его удалении – чрезмерна. В таких случаях, следует воспользоваться архаичным вариантом стандартного заголовка для одно-записных, необъектных данных (STD_HEAder for Non-Object Record, Таб. 5), который позволяет реализовать минимальный жизненный цикл сущности, от создания до удаления, в пределах одной записи.

Таб. 5 STD_HEANOR – Стандартный заголовок необъектной записи

№	Поле	Тип	Описание
01	**ZRID**	bigint	уникальный идентификатор записи этого типа
02	**ZVER**	bigint	номер версии записи
03	ZSID	bigint	идентификатор *области данных*, к которой принадлежит версия записи
04	ZLVL	smallint	уровень секретности записи (используется мандатной СРД)
05	ZPID	bigint	идентификатор ZRID записи parent, в соответствии со структурой БД
06	ZSTA	char(1)	статус версии ∈ {N,C,D} (для более сложных случаев см Таб. 3)
07	ZDATE	timestamp	дата вступления в действие этой версии записи
08	ZUID	bigint	идентификатор пользователя, создавшего версию записи
09	ZDATO	timestamp	время прекращения действия версии записи
10	ZUIDO	bigint	ID пользователя, создавшего версию записи или удалившего запись

2.3.4. Object ID (ZOID) – идентификаторы ОД; ссылки на ОД; ОД-контейнеры

TIS/SQL требует, что-бы все объекты данных и все записи во всех таблицах были снабжены уникальными суррогатными ключами. Это требование обусловлено идеологической установкой на редактируемость любых прикладных данных (в т.ч. ключевых). Главной частью суррогатного ключа любой записи, принадлежащей ОД, является уникальный идентификатор этого ОД – ZOID. Дополнительно к ZOID, каждая запись имеет идентификатор ZRID, однозначно идентифицирующий ее среди других записей того же типа, принадлежащих тому же ОД. Необъектные данные не имеют ZOID, а идентификатор ZRID, для них, является уникальным на множестве всех записей одного типа.

На практике, для получения уникальных идентификаторов рекомендуется использовать генераторы последовательных целых чисел, т.н. SEQUENCE, если они поддерживаются используемой СУБД. Иначе – такой генератор можно реализовать самостоятельно, в виде функции, с использованием вспомогательной таблицы. Обычно используется по одному генератору на каждую таблицу, однако может оказаться целесообразным использовать один генератор для всех ZOID и ZRID, что облегчит управление им, с целью получения идентификаторов, уникальных для нескольких инсталляций системы, эксплуатируемых параллельно, со слиянием всех данных в единой централизованной БД.

Поскольку единственным неизменным на всем протяжении жизни ОД идентификатором является ZOID, именно его, и только его, следует использовать всегда, когда необходимо сослаться на конкретный ОД. Для установления ссылки из одного ОД на другой, в первом из них следует иметь поле, тип данных которого совпадает с типом данных выбранным для ZOID. Обычно, имя такого поля состоит из полного или сокращенного *имени типа ОД*, на который будет установлена ссылка, и аффикса "`_id`" (например: `document_id, doc_id, inventory_id, inv_id`). Если одно и то же поле может хранить ссылки на разные типы ОД – то вместо имени типа, можно воспользоваться каким-нибудь функциональным обозначением или, в самом общем случае – обозначением "`object`" (например: `ref_id, item_id, obj_id`). В большинстве случаев, полессылку следует проиндексировать.

TIS/SQL подход изначально и сознательно не накладывает никаких ограничений на значения полей-ссылок, рассматривая их как обычные прикладные данные. Таким образом, допускаются ссылки на несуществующие ОД; ОД недоступные пользователю, устанавливающему ссылку; ОД неожидаемого типа и т.п. *Контроль качества* содержимого полей-ссылок, для каждого типа ОД, следует проработать и реализовать отдельно, в объекто-специфичных процедурах контроля качества, как ссылающегося, так и целевого ОД. Однако, не следует забывать, что правила по которым осуществляется контроль качества, могут и будут изменяться с течением времени, подстраиваясь под новые потребности. Следовательно – существующие (или существовавшие) в БД версии ОД, могут этим правилам не удовлетворять. Также, следует обратить внимание, что при чтении ОД, содержаще-

го ссылку на другой ОД, пользователь может столкнуться с тем, что последний не разрешен ему СРД, т.е. выглядит как несуществующий. Из выше сказанного следует, что все программное обеспечение должно разрабатываться с учетом возможности любых вышеописанных ситуаций, обеспечивая их корректную обработку и позволяя пользователю работать с любым "некорректными" ссылками.

С самого начала следует выделить и описать два принципиально различных класса ссылок – это «*обыкновенные ссылки»* и «*ссылки-вложения»*. Если первые не накладывают никаких ограничений и ни к чему не обязывают целевой ОД, то вторые требуют что-бы в каждый момент времени целевой ОД был «вложен» только в один объект-*контейнер*, и «принуждают» его перемещаться вместе с этим «контейнером». Примером *обыкновенной ссылки* может служить инвентарная ведомость, со ссылками на ОД, подвергшиеся инвентаризации, а примером *ссылки-вложения*, будет описание ящика-контейнера, с перечислением содержащихся в нем объектов, каждый из которых описан в БД отдельным ОД. Если для *обыкновенной ссылки* можно пренебречь *контролем качества*, не нарушая логики функционирования системы, то для *ссылок-вложений* такой контроль является обязательным. При проектировании ИС, для каждого *поля-ссылки* следует определиться по следующим пунктам:

1. Допустимые типы ОД
2. Возможность ссылок на несуществующие ОД
3. Возможность последующего удаления целевого ОД, без изменения ссылки
4. Какими правами,, по отношению к целевому ОД, должен обладать пользователь, устанавливающий ссылку.
5. Допустимость множественных ссылок на один и тот же целевой ОД из разных ОД
6. Допустимость множественных ссылок на один и тот же целевой ОД из одного ОД
7. Допустимость ссылок на самого себя
8. Допустимость ссылок на ОД в других областях данных.
9. Должно-ли некоторое действие над ОД (например – перемещение), вызывать такое же действие над теми ОД, на которые он ссылается.
10. Какие еще ограничения накладывает ссылка на ссылающийся и целевой ОД, а также на другие ОД и другие поля-ссылки в системе.

Следует быть готовым, при возникновении потребности, реализовать для каждого типа ОД следующие универсальные механизмы обработки ссылок:

- получения списка ZOID всех объектов данных, вложенных в конкретный ОД.
- получения списка всех ZOID, которые ссылаются на конкретный ОД.
- получения списка всех ZOID, на которые ссылается конкретный ОД.
- преобразование всех ссылок в конкретном ОД по таблице соответствия старых и новых ZOID

Первая операция необходима для реализации операции перемещения или передачи ОД со всеми вложениями; вторая – для процедур контроля качества; третья – для четвертой, которая потребуется, если операция перемещения должна быть реализована как удаление всех перемещаемых ОД и заведение их копий, с новыми ZOID. Последнее может потребоваться, если перемещение является передачей информации между двумя инсталляциями системы, с несогласованной генерацией ZOID.

> **NB** Для контроля качества полей-ссылок, особенно вопросов единственности, и для унификации алгоритмов обработки ссылок, возможно использование т.н. *индексных таблиц*, суть которых состоит в поддержании вторичных таблиц, отражающих актуальное состояние некоторой выборки данных на последний момент времени. Детально об *индексных таблицах* будет рассказано в следующем разделе.

2.3.5. Logical Transaction – логическая транзакция

Логическая транзакция, есть согласованная последовательность *физических транзакций*, направленная на изменение текущего, актуального состояния данных в ИС. Все физические транзакции, выполненные между началом и завершением логической, изменяют данные относящиеся к «возможному будущему» состоянию системы. Перенос изменений в «настоящее» состояние (или их отмена), осуществляется одномоментно, в рамках одной физической транзакции, являющейся завершением транзакции логической.

Физическая транзакция – есть обычная транзакция в терминах используемой СУБД, т.е. после-

довательность изменений в БД, которые становятся доступными за пределами сессии, в которой они выполнены, только после успешного завершения SQL оператором COMMIT TRANSACTION.

Потребность во введении понятия *логической транзакции*, проистекает из перехода к по-записному обновлению БД через вызовы хранимых процедур. Поскольку каждый вызов хранимой процедуры должен (или, по крайне мере – может) являться отдельной физической транзакцией, то результаты его выполнения становятся доступными всем остальным сессиям с БД сразу, что не всегда допустимо. Осознание этой проблемы повлияло на формирование понятия *объекта данных*, о котором рассказывалось в разделе 2.3.1, где процедура создания или изменения ОД является типичной логической транзакцией.

Однако применимость этого понятия не ограничивается узкими рамками операций над одним ОД. Как минимум, полезно расширить его, реализовать операцию одновременного «закрытия» нескольких «открытых» ОД таким образом, чтобы одновременно вступили в действие все новые версии, всех указанных документов, или, в случае неудачи, – ни одной. Также, в виде логических транзакций следует реализовывать много-шаговые и много-параметрические операции, над отдельными объектами и их множествами, не укладывающиеся в стандартную парадигму редактирования документа, например – суммирование, вычитание, деление, исполнение и т.п.

Главное качество, привносимое понятием логической транзакции, это возможность привязки к ее завершению, сколь угодно сложного процедурного кода. Две типовые задачи, выполняемые при завершении логической транзакции, это *контроль качества* и коррекция *индексных таблиц*.

Контроль качества (Quality Control, QC) – процедура проверки соответствия предоставленных данных требованиям, наложенным на каждое поле, каждый ОД по совокупности его записей, и все ОД в системе по совокупности. TIS/SQL сознательно и принципиально не использует для обычных таблиц с пользовательскими данными никаких ограничивающих условий (SQL CONSTRAINTS), кроме первичного ключа (PRIMARY KEY), возлагая поддержание согласованного состояния данных на процедурный контроль качества. Отчасти это связано со сложностью формулирования таких условий для таблиц, хранящих одновременно несколько версий записей, но в гораздо большей степени обусловлено потребностями в гибкости, понятности и «линейности», свойственной процедурному подходу, а также – возможностью снабжать отказы вразумительной диагностикой.

Индексная таблица – вспомогательная таблица, постоянной хранящая выборку актуальных, в настоящий момент, данных из других таблиц. На такую таблицу могут быть наложены ограничения (CONSTRAINTS), в соответствии с правилами *контроля качества* для данных, подпадающих под условия выборки, а сами данные могут быть проиндексированы в соответствии с потребностями внутреннего и внешнего, по отношению к ядру системы, ПО. Индексные таблицы предназначены для облегчения процедур контроля качества, ускорения некоторых запросов (особенно – агрегирующих) и универсализации некоторых алгоритмов обработки. Вопросы предоставления доступа к индексным таблицам из прикладного ПО и следующие из этого вопросы разграничения доступа к ним – для каждой подобной таблицы решаются индивидуально.

> **NB** На стыке заполнения *индексных таблиц* и *контроля качества* находится еще одна ценная возможность – к завершению логической транзакции можно привязать процедуру подсчета контрольных сумм новых версий ОД ядром системы, а также, тесно зависимую от этого процедуру электронной цифровой подписи всех новых версий их автором, выполняемую на стороне клиента и проверяемую сервером. Это выводит ИС на совершенно новый уровень «доверенности».

Второе, побочное качество логических транзакций – способность существовать сколь угодно долго, даже переносить разрывы сессий с БД и перезапуск сервера СУБД. При этом, поскольку каждая отдельная физическая транзакция атомарна и выполняется за минимальное время, то логическая транзакция не блокирует доступ к данным других сессий, и не создает предпосылок для deadlock. Обратной стороной этого является существенно меньшая скорость обновления данных, что становится заметно на массовых операциях. Впрочем – здесь есть простор для оптимизации.

2.3.6. Scope – область данных

Область данных (scope) – совокупность разнотипных данных, принадлежащих к одной категории, с точки зрения разграничения доступа. Для распределенных и реплицируемых систем, *область данных* является той единицей, которая целиком храниться и контролируется одним узлом (инсталляцией) системы, и всегда реплицируется на другие узлы целиком, поскольку *объекты данных*, принадлежащие ей, находятся между собой в сложных отношениях взаимной зависимости.

Любые прикладные данные, всегда принадлежат какой-либо *области данных*, причем в каждый конкретный момент времени – только одной. Однотипные данные, принадлежащие разным областям данных, хранятся в одних и тех же таблицах. Количество областей данных не ограничено, и заранее не определено, а их создание и настройка – является штатной операцией по управлению ИС.

Ограничения, накладываемые на прикладные данные, могут основываться на их принадлежности к области данных. Например, условие уникальности значения некоторого поля, может быть ослаблено с «уникально среди всех ОД этого типа в настоящий момент», до «уникально среди всех ОД этого типа, находящихся в той же области данных, в настоящий момент». Естественно, приведенные пример – простейший, сложность и комплексность таких условий ограничена только потребностями решаемой задачи и здравым смыслом.

Основное, но не единственное, применение областей данных – реализация в пределах одной БД, на общей структуре таблиц ОД, нескольких подчиненных БД, идентичных по структуре, но с независимой политикой разграничения доступа для каждой из них. При этом их данные являются частью общего массива и одновременно доступны любому пользователю системы, имеющему соответствующие права доступа. В процессе существования системы, эти подчиненные БД могут быть выделены в отдельные инсталляции системы, с возможностью репликации сделанных изменений в централизованное хранилище и/или между собой.

Эволюция понятия область данных начиналась с некоторой разновидности объектов, являющихся своеобразными «супер-контейнерами», прямо или косвенно содержащих все остальные объекты. Следующим шагом стало осознание, что разграничение доступа ко всем объектам в системе, следует осуществлять основываясь на том, в каком «супер-контейнере» они находятся. Замечу, что происходило это до формулирования понятия *объект данных*, а структура БД находилась в состоянии близком к состоянию 2 на Рис. 3. Таким образом, реализованная система разграничения доступа оказалась состоящей из иррегулярных SQL условий, обслуживающих разные таблицы, и что самое неудобное – жестко связанной с несколькими специфическими понятиями прикладной области. В дальнейшем иррегулярность являлась постоянным источником ошибок, а связанность всех понятий СРД с прикладной область – мешала развитию и той и другой.

Радикальным решением всех осознанных проблем и стала абстрактная концепция *области данных*, как четко очерченной и поименованной их совокупности, используемой при назначении прав доступа, и как некоторое минимальное подмножество ОД, с которым можно работать изолированно. Таким образом, прикладная область разрабатывается и развивается отдельно, а для целей разграничения доступа описывается в терминах *областей данных*, причем последнее может быть сделано при развертывании системы, ее администратором, без участия разработчиков.

Наиболее естественных способом разграничения доступа в TIS, является разрешение выполнять определенную операцию (чтение,запись,создание,удаление), по отношению ко всем ОД определенного типа в указанной области данных. Для целей разграничения доступа, в каждой записи прикладных данных, присутствует специальное служебное поле, `ZSID` – идентификатор области данных. В системных таблицах, поле, ссылающееся на область данных, обычно называется `sid`.

Англоязычный вариант термина, «*scope*», был подобран как наиболее подходящий к описываемому явлению большинством своими значений и их оттенками, при этом наименее «заезженный» в сфере информационных технологий. Немаловажным оказалось и созвучие славянскому корню «скоп» в смысле «скопление», коим по сути и является множество ОД, принадлежащих одной *области данных*. Таким образом, слово «*скоп*», является очевидным кандидатом в русскоязычные синонимы для «*области данных*», и как русская транскрипция для «scope», и по своему смыслу.

2.3.7. Subsystem – подсистема

Подсистема – совокупность типов ОД и приложений для их обработки, предназначенные для обслуживания потребностей отдельной задачи или отдельной категории пользователей.

В типовой корпоративной ИС, подсистемами являются бухгалтерия, кадры, учет основных фондов и т.п. TIS подход полагает, что данные всех подсистем хранятся в одной БД, с возможностью установления взаимных ссылок и анализа всей информации о предприятии по совокупности. Механизм *областей данных* позволяет эффективно разграничить доступ к информации различных подсистем, даже если какие-то типы ОД, или даже – сами ОД, у них окажутся общими.

Каждая подсистема может разрабатываться и развиваться достаточно независимо от остальных, что создает хорошую почву для внутренней параллельности процесса разработки. В любой TIS/SQL системе будет минимум две подсистемы – это СРД и работа с прикладными данными.

2.3.8. Transit – перемещение и переходное состояние ОД

Все данные в системе принадлежат к какой-либо *области данных*, при этом в каждый конкретный момент времени – только одной. Естественно, встает проблема перемещения ОД из одной области в другую, и решение ее не так просто, как кажется на первый взгляд. Вот список вопросов, на которые следует ответить, приступая к реализации операции «перемещение»:

1. Каков алгоритм изменений записей в таблицах БД, результатом которого будет перемещение ОД из одной области данных в другую.
2. Как эта операция согласуется с версионностью объектов данных.
3. Что происходит с историей изменений перемещенного ОД.
4. Должна-ли операция перемещения быть одно-шаговой – `перемещение(что,куда)`, или двух-шаговой – `{отправить(что, куда), принять(что) }`.
5. Какими правами должен обладать пользователь, совершающий одно-шаговую операцию `перемещение(ОД, в область данных)`, по отношению к этому ОД; к области данных, в которой он находится; и к области данных, в которую осуществляется перемещение.
6. Какими правами должны обладать пользователи, по отношению к перемещаемому ОД и двум областям данных, для выполнения операций `отправить()` и `принять()`, при двух-шаговом перемещении.
7. Где находится ОД, между операциями `отправить()` и `принять()`. Кто его может просматривать в это время.
8. Что должно происходить с ОД, вложенными в перемещаемый.
9. Что должно происходить с *обыкновенными ссылками* перемещаемого ОД, и ссылками на него самого из других ОД.
10. Как организовать передачу ОД между областями данных, размещенными в разных БД.

Ответы на эти вопросы будут зависеть от конкретной решаемой задачи, и, в большинстве систем, будут различными для разных подсистем или даже для разных типов ОД, в пределах одной подсистемы. Выбор конкретного технического решения – зависит от потребностей прикладной задачи, но проектировать его следует отталкиваясь от группы идей и шаблонов приводимых далее.

Наибольшая определенность есть по первым трем вопросам – перемещение должно корректным образом вписываться в последовательность версий ОД, не нарушать ведение истории изменений и обеспечивая воспроизводимость состояния БД на том отрезке времени, когда ОД принадлежал старой области данных. Осуществляется это следующим образом:

1. Выполняется операция аналогичная удалению ОД:
   1.1. Действующая версия помечается как устаревшая со ZSTA="M" (moved);
   1.2. Действующие версии записей ОД, помечаются как прекратившие свое существование вместе с «перемещаемой» версией. Это сохраняет их в старой области данных, для воспроизводимости ее состояния на любой момент времени в прошлом.
2. Выполняется операция по созданию новой версии ОД в новой области:
   2.1. Создается следующая версия ОД, ZSTA='N', ZSID=<новая область данных>
   2.2. Создаются новые версии (копии) всех записей, существовавших в предыдущей версии, при

этом им присваивается ZSID новой области.

3. Выполняется процедура контроля качества ОД, с учетом его нового положения, и если он успешен – новая версия ОД становится действующей. Иначе – следует перейти к редактированию этой новой версии ОД для устранения возникших ошибок.

По этой схеме, разные версии ОД могут принадлежать разным областям данных, и именно ее следует считать базовой. Разновидностью этой схемы будет удаление ОД в одной области, и создание его копии в другой, но уже с новым ZOID. При такой схеме передачи, следует организовать учет соответствий старых и новых ZOID, для целей анализа перемещений в спорных ситуациях.

В некоторых специфических системах вся ценность документа состоит именно в непрерывной истории его изменений, а воспроизводимость истории с точностью до принадлежности каждой записи области данных – неважна. В этом случае допустимо реализовать передачу ОД в другую область вместе со всей историей, что реализуется коррекцией ZSID всех версий, всех его записей. Такой же вариант операции необходим, если ОД должен быть изъят из области данных вместе со всей своей историей, при изменении политики разграничения доступа, после которого пользователи, работающие с областью и ее историей, не должны иметь доступ к старым версиям перемещенного ОД. Для сложной, долгоживущей системы такую операцию следует реализовать обязательно, в предположении, что ее потребителем будет системный администратор, перед которым встанет задача реорганизации структуры *областей данных*. При этом следует понимать, что такое «перемещение» плохо совместимо с распределенными и репликативными системами.

Следующий вопрос – одно-шаговая или двух-шаговая операция – полностью зависит от семантики реального явления, стоящего за операцией смены области данных. Двух-шаговая операция больше подходит для большинства целевых систем, в которых перемещение должно быть штатной операцией, доступной ординарным пользователям. Пересечение границ областей данных в таких системах – это чувствительная операция, за которую должны «расписаться» два человека, с двух сторон (передающей и принимающей). В этом контексте, одно-шаговое перемещение можно рассматривать как разновидность двух-шагового, при котором один и тот же пользователь имеет право «расписаться» за обе стороны.

При двух-шаговом перемещении, возникает промежуток времени, когда ОД отправлен, но еще не принят, т.е. находится в *транзите*. Это особое состояние требует тщательной идеологической проработки, в результате которого должны быть сформулированы правила обращения с ОД, находящимися в таком переходном состоянии. Исходная установка такова – ОД, находящиеся в транзите, не должны редактироваться, а информация о них должна быть доступна и отправителю, и получателю. В TIS/SQL, для учета информации об ОД в транзите, предлагается использовать специальную таблицу `ZTransit`, а для просмотра информации о них – специальные фильтрующие view, учитывающие права пользователей на передачу и прием информации между областями данных.

Что касается прав пользователей на перемещение, то здесь следует отталкиваться от механизма *привилегией* (capabilities), рассматриваемом в разделе посвященном СРД. Для базовых операций перемещения между областями достаточно следующих трех привилегий:

1. MOVE_WITH_HISTORY – глобальная привилегия, позволяющая перемещать любой ОД, из любой области в любую область данных, вместе со всей историей изменений.
2. MOVE_SEND – привилегия на область данных, позволяющая отсылать любой ОД из этой области в любую другую и отзывать из транзита ранее отосланные, но еще не принятые.
3. MOVE_RECEIVE – привилегия на область данных, позволяющая принимать любые ОД, отосланные в эту область из любой другой, а также – отказывать в приеме.

По поводу вложенных ОД – в общем случае, они должны последовать за объектом-контейнером. Для этого надо их все обнаружить, и поместить в транзит, установив связь с контейнером самого верхнего уровня, для упрощения последующих операций приема и отзыва. Прием такого вложенного объекта возможен только вместе с приемом контейнера самого верхнего уровня и всеми его вложениями, иначе операция становится крайне нетривиальной, и требует отдельной проработки.

Последние два вопроса не имеют универсальных ответов. Правила обработки *обыкновенных ссылок* полностью зависят от потребностей прикладной области, а детали передачи информации между различными инсталляциями системы – выходят за рамки этого документа. Тем не менее,

сформулированы они здесь сознательно, о них следует помнить, и быть готовыми к их решению.

В заключение, следует описать еще две ситуации, которые могут описываться понятием транзита, хотя и не являются передачей ОД между областями данных:

- Первая – изменение уровня секретности ОД. Скорее всего, осуществляться оно должна как двух-шаговая операция – запрос на изменение уровня секретности и его утверждение, между которыми ОД доступен для чтения, но недоступен для редактирования.
- Вторая ситуация, это некоторое положение объекта, в котором его качества постоянно изменяются во времени, по некоторому известному или неизвестному закону. Получается, что мы не можем доверять информации, содержащейся в ОД, или должны вносить в нее поправку, исходя из времени проведенного в этом состоянии и прочих условий. Наглядным примером такой ситуации служит полено в печи. Для ее описания также подходит концепция *транзита*.

2.3.9. SAM – система разграничения доступа

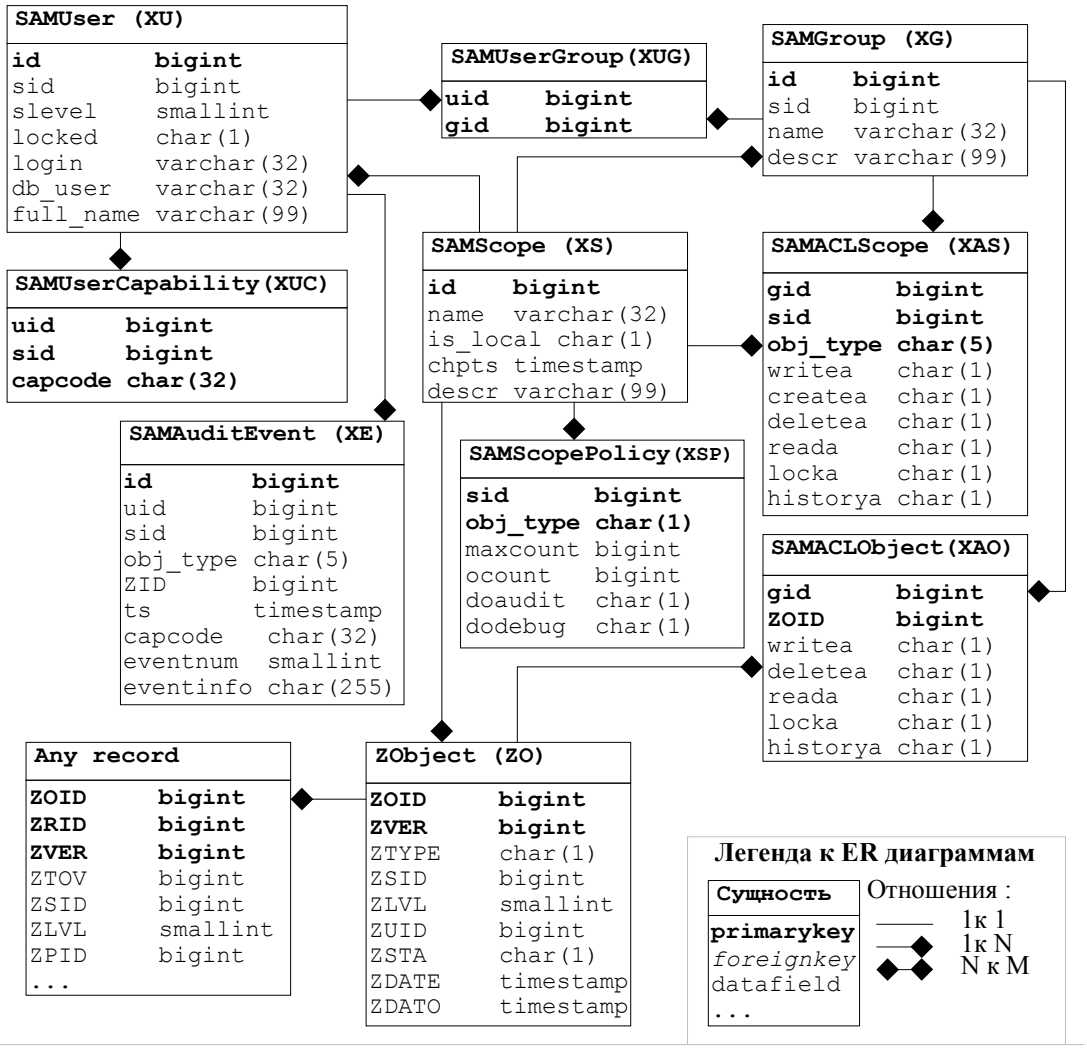


Рис. 4 ER диаграмма СРД (SAM)

Первые два «кита» TIS/SQL подхода, две базовых абстракции *объект данных* и *область данных*, понятия весьма «сектантские» (т.е. оригинальные и специфичные для TIS); при этом их формулировки – компактны, а сами они – «атомарны» по назначению. Последний, третий «кит», *Система Разграничения Доступа* (*СРД*), наоборот – обширный комплекс, состоящий из множества подобных «атомарных» абстракций, в т.ч. и двух вышеназванных. Однако, все эти новые составные части уже не являются чем-то оригинальным, все они хорошо знакомы любому, касавшемуся проблем защиты информации в компьютеризированных системах. Ценность последнего из «китов»

не во введении какого-то нового понятия, а в том, чтобы показать, каким образом уже существующие могут (должны) работать совместно, для решения поставленных задач.

Англоязычная версия названия подсистемы СРД – SAM, есть аббревиатура от Security Access Management, совпадающая с аббревиатурой сходной подсистемы Windows NT (в которой она образована от наименования «Security Accounts Manager»). Далее, оно используется в качестве аффикса, при образовании имен всех сущностей, относящихся к подсистеме СРД.

Прежде чем приступать к детальному описанию конструкции SAM, представленной на Рис. 4, – следует оговориться, что предлагаемая реализация не является абсолютом. Скорее ее следует рассматривать, как работоспособный прототип и некоторую базовую, живую форму – своеобразный кладезь идей-генов, из которого должна эволюционировать СРД, подходящая к условиям конкретной ИС. Абсолютом TIS/SQL являются: абстрагированность СРД от прикладной части; инвариантность прикладного программного обеспечения, по отношению к деталям ее реализации и настройкам. В результате, СРД может меняться и перестраиваться, а накопленный массив данных и прикладное ПО – оставаться неизменными, равно как и представление о них программистов и пользователей.

Построение СРД начинается с конкретизации двух абстрактных понятий – «*субъект доступа*» и «*объект доступа*». В TIS, субъектами доступа, всегда являются *пользователи.* Каждый из них описан отдельной записью в таблице `SAMUser`. Пользователем является не только человек, но и любая программа-сервис (робот), с точки зрения СРД – принципиальной разницы между ними нет.

С *объектами доступа* ситуация несколько сложнее – обычно, объектом доступа принято считать каждый отдельный *файл*, но в TIS понятия *файла* нет, зато есть очень близкое к нему понятие *объект данных*. Тем не менее, попытка объявить объектом доступа ОД и построить СРД по аналогии с файловой системой, где права доступа назначаются на каждый ОД индивидуально, обнаруживает ряд проблем, показывающих, что такой способ разграничения доступа может быть одновременно чрезмерно избыточным и недостаточным.

Первая проблема – это разрешения на создание ОД. Файлы создаются в директориях, которые суть есть те же файлы, а следовательно для них можно, до некоторой степени, применять те же принципы назначения прав доступа (право писать директорию – право создавать в ней файлы). ОД существуют сами по себе, вложение одного ОД в другой – необязательно. Здесь очевидна недостаточность файлового принципа.

Вторая проблема – назначение прав доступа на части ОД, такие как типы записей внутри ОД, конкретные записи и поля в записях. Для файлов такой проблемы нет, поскольку файл это просто последовательность байт, в то время как ОД – жестко структурированный документ.

Третья проблема состоит в том, что назначение и поддержание в правильном состоянии прав доступа для каждого из десятков или сотен тысяч документов – является чересчур трудоемкой задачей для администратора безопасности – если он будет относиться к это задаче внимательно, и неизбежно приведет к ошибкам – если проявит небрежность. Есть некоторый предел объема информации, с которым способен работать человек. При его превышении – он бессознательно производит сокращение и упрощение, что приводит к потерям некторых деталей, а в задачах защиты информации от этих деталей может зависеть все. Здесь очевидна избыточность файлового принципа. (Кстати, эта же проблема свойственна большим, долгоживущим файловым архивам. Администратор любого такого архива, при внимательном рассмотрении, найдет массу ошибок в уже назначенных правах доступа.)

Четвертая проблема, обусловленная наложением файлового принципа на SQL основу, – неоправданные потери производительности, которые может вызвать обработка массива прав доступа, сопоставимого по объему с массивом прикладных данных.

Для решения этих проблем, следует оглянуться назад и вспомнить принципы разграничения доступа, реализованные в большинстве СУБД, и встроенные в стандарт SQL, *объектом доступа* в которых является вся таблица. Для многих задач этого оказывается достаточно. Задачам, для которых создавался TIS/SQL, не хватало буквально самой малости – возможности организовать в пределах одной таблицы нескольких подмножеств, для которых можно было-бы назначать права доступа индивидуально, и композитной структуры данных, для хранения документов, более сложной чем отдельная строка в реляционной таблице. Именно для устранения этих недостатков введены понятия «*область данных*» и «*объект данных*».

В результате, основным *объектом доступа* TIS/SQL, является *область данных*. Права доступа (создавать/писать/читать/удалять) – назначаются на *типы объектов данных* в *области данных*. Если возникает необходимость, то для некоторых или всех *типов записей* (таблиц), принадлежащих *типу ОД*, внутри *области данных,* – права могут назначаться индивидуально. То же самое касается и конкретных полей, находящихся в этих *типах записей*. При этом, конкретный ОД также остается *объектом доступа*, а при *аудите* действий пользователей – это главная контрольная точка, но акцент в разграничении доступа смещен выше – от самих документов к их категориям (областям данных), что напоминает мандатный принцип разграничения доступа.

Возможно (и даже, возможно, – необходимо), реализовать механизм назначения прав доступа на отдельные ОД и их части (типы записей и поля), но его применимость в реальных системах весьма ограничена – не следует им злоупотреблять (по причинам указанным выше).

Резюмируя вышесказанное:

- TIS подход имеет иерархическую систему подчиненности (вложенности) *объектов доступа* – область данных{тип ОД{тип записи{поле в записи}}} и объект данных{тип записи{поле в записи}}.
- Система назначения прав доступа – разрешительная (т.е. то что не разрешено – запрещено).
- Права доступа, назначенные на *объект доступа,* распространяются и на подчиненные ему *объекты доступа*, если политика разграничения доступа не требует для них индивидуальных разрешений.

Возвращаясь к ER-диаграмме (Рис. 4), следует сделать ряд пояснений, облегчающих ее понимание. На диаграмме показаны только важные для понимания связи и отношения. Все показанные отношения 1:N подразумевают некоторую идеологическую подчиненность множества сущностей N сущности 1. Чтение диаграммы следует начинать с верхнего левого угла, с сущности SAMUser, представляющей «*субъект доступа»*.

- Первое направление чтение – по часовой стрелке SAMUser, SAMUserGroup (членство в группах доступа), SAMGroup (Группы доступа), SAMACLScope (ACL *группы доступа* на *области данных*), SAMACLObject (ACL на *объекты данных*). Этот граф охватывает стандартную схему назначения прав доступа к данным (права на создание, редактирование, чтение, удаление и т.п.).
- Второе направление (против часовой стрелки) – SAMUser, SAMUserCapability (привилегии пользователя) и SAMAuditEvent – описывает назначение прав доступа, не укладывающихся в стандартную схему и аудит действий *субъекта доступа*.
- И наконец, в центре диаграммы, находится «паук», протянувший свои «лапы» ко всем остальным сущностям – SAMScope (*област данных*, см. Таб. 6). Причем, часть этих связей просто не показана, что-бы не загромождать рисунок. На диаграмме всего две сущности, не содержащие в себе явной ссылки на область данных (поля sid или ZSID) – SAMUserGroup и SAMACLObject. Таблица SAMScopePolicy (см. Таб. 7) предназначена для перечисления типов ОД, разрешенных к созданию в конкретной *области данных*.

Далее, все сущности SAM, представленные на ER-диаграмме (Рис. 4), будут описаны и рассмотрены детально, с указанием способов применения и направлений развития.

Таб. 6 SAMScope (XS) – Область данных

#	Поле	Тип	Название	Описание
01	**id**	bigint	Идентификатор	Уникальный, неизменяемый числовой идентификатор
02	name	varchar(32)	Название	Уникальное имя области данных
03	is_local	char(1)	Размещена локально	Область данных управляется этой инсталляцией системы
04	chpts	timestamp	Дата актуальности	Для нелокальных областей – дата/время, на которые актуально состояние данных, для локальных – любая следующая логическая транзакция в этой области, должна иметь дату/время завершения позже chpts (checkpoint timestamp)
05	descr	varchar(99)	Описание	Свободное описание

Таб. 7 SAMScopePolicy (XSP) – Политика область данных

#	Поле	Тип	Название	Описание
01	**sid**	bigint	Область данных	Идентификатор области данных, к которой относится эта запись политики
02	**obj_type**	char(1)	Тип ОД	Тип ОД, разрешенный к созданию в этой области
03	maxcount	bigint	Max кол-во	Максимально допустимое кол-во ОД этого типа, которое разрешено создать
04	ocount	bigint	Текущее кол-во	Текущее кол-во ОД этого типа в этой области (поддерживается автоматически)
05	doaudit	char(1)	Выполнять аудит	Выполнять аудит всех значимых действий над ОД этого типа
06	dodebug	char(1)	Отладочный аудит	Выполнять более детальный аудит, полезный только для отладочных целей

2.3.9.1. SAMUser – пользователь

Таб. 8 SAMUser (XU) – Пользователь

#	Поле	Тип	Название	Описание
01	**id**	bigint	Идентификатор	Уникальный, неизменяемый числовой идентификатор
02	sid	bigint	Область данных	(зарезервировано) ID области данных, в которой находится этот пользователь
03	slevel	smallint	Уровень доступа	Уровень доступа к информации (для мандатной СРД)
04	locked	char(1)	Доступ блокирован	Доступ пользователя в систему заблокирован
05	login	varchar(32)	Login	Уникальное регистрационное имя пользователя системы (login)
06	db_user	varchar(32)	Пользователь БД	Пользователь БД, соответствующий этому пользователю системы
07	full_name	varchar(99)	Полное имя	Полное имя пользователя (ФИО)

Таб. 9 SAMUserGroup (XUG) – Участие в группе доступа

#	Поле	Тип	Название	Описание
01	**uid**	bigint	Пользователь	ID пользователя, включенного в группу доступа
02	**gid**	bigint	Группа доступа	ID группы доступа, в которую включен пользователь

Все *субъекты доступа* должны быть описаны в системе, в таблице `SAMUser`, с присвоением каждому уникального числового идентификатора uid (поле `SAMUser.id`). Именно этот числовой идентификатор используется всегда, когда необходимо сослаться на данного субъекта. Помимо числового идентификатора, каждый пользователь имеет уникальный символьный идентификатор-имя (SAMUser.login), который обычно используется при отображении документов, вместо uid. Поле id должно оставаться неизменным, а вот login, как и все остальные поля, можно откорректировать.

Таблицы SAM не поддерживают ведения истории, поэтому для учета любых изменений в них следует использовать журнал аудита (`SAMAuditEvent`). Особое внимание при этом следует уделить таблице SAMUser, поскольку это главное и единственное описание, что представляет собой *субъект доступа,* стоящий за uid. По этой же причине, вместо удаления учетной записи пользователя, следует применять ее блокировку (см. поле `SAMUser.locked`), что обеспечивает корректное преобразование существующих uid в login/full_name. Обычное удаление – для очень специфических случаях.

Основные права доступа к данным, такие как чтение/запись/создание/удаление, пользователи получают через участие в *группах доступа*. Более специфические права – назначаются индивидуально, через механизм *привилегий* (SAMUserCapability). Впрочем, при необходимости, допустимо реализовать назначение *привилегий* через *группы доступа*. Строго персонально назначаются только метки и уровни доступа в мандатной СРД (см slevel).

Поле sid, в описании пользователя, задает еще одно направление, в котором может развиваться СРД. Вместо плоского списка пользователей системы, управляемого одним администратором безопасности, можно организовать несколько таких списков, каждый из которых может управляться отдельным сотрудником. Вот одна из идей, пригодная для сложной системы, обслуживающей

несколько подразделений с большой текучкой кадров:

1. Для каждого подразделения выделяется *область данных*
2. В этой *области данных* создается одна или несколько *групп доступа*.
3. Руководителю подразделения (или лицу им назначенному) предоставляются полномочия заводить новых пользователей, в этой области данных, а также включать их в группы доступа, принадлежащие той же области. При этом изменять полномочия этих групп – ему не позволяется.

Значительно более сложная конструкция будет необходима при построении распределенных и репликативных систем. Детали ее реализации могут существенно варьироваться, но главная идея – действовать в системе могут только пользователи, принадлежащие локальным областям данных, а при репликации состояния области данных – должен реплицироваться и список пользователей, описанных в ней.

Развивать СРД можно и в сторону упрощения. Например, в системах предназначенных для хранения индивидуальных профилей пользователей, где каждый такой профиль представляет собой отдельную область данных, можно использовать поле sid для указания этой области, а вместо включения пользователя в группы доступа – считать что он обладает некоторыми априорными правами в «своей» области. Впрочем, в этом случае можно вообще приравнять понятие *пользователя* к понятию *области данных*, и, либо использовать один и тот же id для обоих, либо – слить два понятия в одном (SAMUser = SAMScope = SAMUserProfile).

За привязку пользователей системы к пользователям БД отвечает поле db_user. По этому соответствию пользователь БД автоматически распознается как конкретный пользователь системы. Полю db_user можно присвоить NULL, указав тем самым, что ни один пользователь БД не должен автоматически получать полномочия этого *субъекта доступа*. Такая учетная запись не является заблокированной – сервер приложений, имеющий привилегию становиться любым пользователем (TRUSTED_USER), может назначить своей сессии этого *субъекта доступа*. В приводимом примере, db_user имеет символьный тип данных, однако предпочтительнее использовать числовые идентификаторы (когда СУБД позволяет работать с ними).

Описательная часть пользователя состоит всего из одного поля – full_name. При желании, можно дополнить SAMUser и другими полями, как то «должность», «телефон» и т.п. Однако не следует превращать SAMUser в кадровую подсистему, используйте для этого нормальный механизм *объектов данных*, обеспечивающий полноценное ведение истории изменений, и позволяющий работать с описаниями любой сложности. Если же все сотрудники, учтенные в кадровой подсистеме, должны быть и пользователями системы – пусть SAMUser станет *индексной таблицей*.

2.3.9.2. SAMGroup – группа доступа

Таб. 10 SAMGroup (XG) – Область данных

#	Поле	Тип	Название	Описание
01	**id**	bigint	Идентификатор	Уникальный, неизменяемый числовой идентификатор (gid)
02	sid	bigint	Область данных	(зарезервировано) ID области данных, в которой находится группа доступа
03	name	varchar(32)	Название	Уникальное имя группы доступа
04	descr	varchar(99)	Описание	Свободное описание группы доступа

Группа доступа – это именованная совокупность прав доступа, которая планируется и составляется администратором безопасности под определенную категорию пользователей или под определенную функцию. Права группе доступа назначаются через механизм ACL.

Пользователь может быть включен в несколько *групп доступа*, в этом случае его полномочия – есть сумма полномочий всех этих групп. Назначение и соотношение полей id и name – аналогично id и login в SAMUser соответственно.

Группы доступа это аналог групп пользователей, но используемый термин подчеркивает смещение акцентов с группирования *субъектов доступа* к группированию полномочий по некоторому функциональному критерию. *Группы доступа* никогда не являются *субъектами доступа*, ссылки на них никогда не встречаются ни в самих прикладных данных, ни в метаданных, их обслуживающих (стандартный заголовок и т.п.). Фактически, понятие *группы доступа* абсолютно локально для подси-

стемы SAM, что отличает его от понятия «*группа пользователей*», которое во многих реализациях является разновидностью или подобием понятия «*пользователь*».

Пример использования зарезервированного поля sid, приведен в предыдущем разделе, посвященном SAMUser, как и некоторые другие идеи, касающиеся использования *групп доступа*. Однако, приводимая реализация является почти предельной, понятие *группа доступа* является внутренним мостом между пользователями и ACL, что не допускает его развития во что-то иное.

2.3.9.3. SAMACL – список контроля доступа

Таб. 11 SAMACLScope (XAS) – ACL на типы объектов доступа в области данных

#	Поле	Тип	Название	Описание
01	**gid**	bigint	Группа доступа	Идентификатор группы доступа
02	**sid**	bigint	Область данных	Идентификатор области данных
03	**obj_type**	char(5)	Тип объекта доступа	Код типа ОД, записи или поля.
04	writea	char(1)	Запись	Право писать в сущ. объект этого типа в этой области
05	createa	char(1)	Создание	Право создавать новый объект этого типа в этой области
06	deletea	char(1)	Удаление	Право удалять объект этого типа из этой области данных
07	reada	char(1)	Чтение	Право читать любой объект этого типа в этой области
08	locka	char(1)	Блокировка	Право устанавливать блокировку на ОД этого типа в обл.
09	historya	char(1)	Чтение истории	Право читать предыдущие версии объектов этого типа в обл.

Список контроля доступа (Access Control List, ACL) – вариант реализации *дискреционного механизма разграничения доступа*. В TIS/SQL подходе, для ACL, используется табличный механизм описания полномочий *групп доступа* по отношению к *объектам доступа*. ACL являются основным, и в большинстве реализаций – единственным источником информации о правах доступа, на основании которого фильтрующие view осуществляют отображение информации для конкретных *субъектов доступа*. Хранимые процедуры редактирования данных, для проверки полномочий, также пользуются информацией содержащейся в ACL. Механизм ACL покрывает от 80% до 99% потребностей в разграничении доступа.

Структура ACL допускает большое разнообразие реализаций. При этом можно одновременно, для одних и тех же данных, использовать несколько видов ACL, что демонстрирует присутствие на ER-диаграмме SAM (Рис. 4) двух таблиц – SAMACLScope и SAMACLObject, на примере которых ведется дальнейшее раскрытие этой темы. Число видов ACL неограничено, но для SQL реализации следует всегда учитывать проблему эффективности использования таких таблиц в SQL запросах фильтрующих view. Общая рекомендация и для проектирования таблиц ACL, и для заполнения их записями – постараться свести и количество таблиц, и записей в них к минимуму. Идеальным вариантом был-бы ACL состоящий из одной таблицы, в котором для каждой пары (субъект доступа, объект доступа) существовало-бы не более одной записи, что позволило-бы строить фильтрующие view на только внутренних объединениях (INNER JOIN), без применения условия EXISTS.

В рассматриваемом случае, все ACL создаются и редактируются только администратором системы, или пользователем, которому предоставлена специальная привилегия (SAM_MANAGE или SAM_MANAGE_ACL). Т.е. обыкновенные пользователи на содержимое ACL повлиять не могут, что существенно отличается от типовой реализации дискреционного способа разграничения доступа, где пользователь создав *объект доступа* – становится его владельцем, а будучи владельцем – может изменять ACL *объекта доступа* по своему усмотрению. В TIS, понятие владельца, наравне с понятием суперпользователь, – признано вредным. Причина такого решения объясняется на следующем примере:

- Пусть пользователь получает право создавать документы типа «X».
- пользователь создает такой документ и становится его владельцем.
- пользователя переводят на другую работу, не предусматривающую доступ к документами «X».
- администратор безопасности лишает его этого права.
- но при этом пользователь сохраняет контроль над теми документами «X», которые он уже создал, поскольку является их владельцем.

Очевидно, что это ненормальная ситуация для нормальной корпоративной ИС. Отсутствие понятия «владелец» не возведено в догму, в отличии от запрета на «суперпользователя». TIS подход акцентирует внимание на проблемы, порождаемые понятием «владелец», и позволяет строить системы без его применения, но окончательный выбор – за проектировщиком конкретной ИС.

Основным типом ACL, на основе которого предлагается строить СРД для типовой ИС – это ACL на *область данных* (`SAMACLScope`, Таб. 11). Данный ACL приписывается группе доступа (`gid`), и позволяет задать разрешения для 6-ти стандартных операций над *объектами доступа* указанного типа (`obj_type`), принадлежащими к области данных (`sid`). Таким образом, *субъект доступа*, для успешного выполнения операции создания ОД типа Document (код типа – "D"), в области данных ScopeX, должен быть включен в некоторую *группу доступа*, имеющую запись SAMACLScope:

```
{gid=<id группы>, sid=<id ScopeX>, obj_type="D", createa="Y",...}
```

То же самое касается 5-ти оставшихся операций над этим (или любым другим) типом ОД. Аффикс "a", в именах полей writea/createa/reada/..., происходит от слова "access". Допустимые значения для этих полей: {"Y" – *разрешено*, "-" – *не определено*, "D" – (зарезервировано) *запрещено*}.

В общем случае, *субъект доступа* имеет право на выполнение некоторой операции, если эта операция *разрешена* в любом ACL одной из *групп доступа*, к которым он принадлежит, и не *запрещена* ни для одной из них.

Реализация *запрета*, имеющего больший приоритет, чем *разрешение*, – связано с дополнительными накладными расходами при выполнении всех SQL запросов. В большинстве систем это не является жизненно необходимым, и им можно пренебречь.

Поле obj_type должно содержать *код типа объекта доступа*, на который распространяется действие данной записи ACL. Размер в 5 символов задан из предположения одно-символьных кодов типа ОД, двух или трех-символьных кодов таблиц (код таблицы = код ОД + 1 или 2 символа) и 4-х или 5-ти символьных кодов полей (код поля = код таблицы + 2 символа). В предельном случае, каждый тип ОД, каждый тип записи (таблица) и каждый тип поля – должны иметь запись в ACL каждой *группы доступа*, которой разрешено с ними работать, – это чересчур расточительно и довольно бессмысленно.

На практике, сразу осуществлять защиту всех полей индивидуально – очень расточительно. Проверка прав доступа при чтении требует замены самого поля на функцию от него, что препятствует использованию индексов, построенных по этому полю. Раздувание ACL – не только ухудшает производительность, но затрудняет их понимание. С учетом использования кодогенератора для всех хранимых процедур и view, обслуживающих каждый тип ОД, добавление индивидуальной защиты для некоторой таблицы или поля – процедура рутинная и тривиальная. Доработка ACL администратором безопасности – займет больше времени и умственных усилий.

В силу вышесказанного, следует начинать с реализации, где права доступа для всех типов записей внутри ОД – равны правам доступа на сам ОД, а права на *типы полей* – равны правам на *типы записей*. Такая реализация проста для понимания, ACL становятся компактными (число записей для каждой группы – не больше числа типов документов), а производительность запросов – максимальной. По мере уточнения потребностей, можно вводить индивидуальную защиту для некоторых типов записей внутри ОД, и некоторых типов полей внутри записей. В крайнем случае, можно начать с реализации, в которой каждый типа записи (таблица) – требует отдельной записи ACL.

Для начальной реализации ACL, в которой нет отдельных прав для редактирования записей, следует принять, что право «создавать» - позволяет создать новый ОД, в то время как право «писать» – позволяет открывать существующие ОД. При любых действиях по редактированию «открытого» состояния ОД (как то: создание/удаление/изменение записей) – достаточно любого из этих прав. Такой подход обеспечит возможность отдельно контролировать доступ к созданию новых ОД и отдельно – к редактированию существующих.

Второй типа ACL, `SAMACLObject`, позволяет осуществлять разграничение на уровне конкретных ОД. Его структура не имеет принципиальных отличий от рассмотренного выше `SAMACLScope`, просто вместо идентификатора области `sid`, у него идентификатор ОД - `ZOID`. Для предлагаемой начальной реализации из его структуры исключены поля `obj_type` и `createa`, но для осуществления разграничения на уровне таблиц и полей – их потребуется вернуть на место. Впрочем, при проектировании реальной ИС – следует рассмотреть возможность полного отказа от этого типа

ACL. В базовой структуре таблиц СРД, на Рис. 4, `SAMACLObject` подчеркивает возможность одновременного использования нескольких структур ACL, и демонстрирует принципиальную возможность разграничения на уровне конкретных документов, эквивалентную встроенной в большинство файловых систем.

Возвращаясь к идеальному, с точки зрения производительности, ACL – следует описать некоторую идею, которая способна привнести в систему новое качество сразу в нескольких сферах: разграничение доступа, аудит и переключение между субъектами доступа в пределах одной сессии с БД. *Идея динамических таблиц СРД* состоит в том, чтобы определять пользователя системы и проверять права доступа не по самим таблицам SAM, содержащим статичный набор данных, а по таблицам, содержащим его динамическое подмножество, порождаемое по запросам со стороны пользователя. Таким образом, пользователь БД, перед началом работы должен будет объявить, от имени какого субъекта доступа он будет работать, и с какими данными (например – задать список *областей данных*). Это позволит сгенерировать для него индивидуальный ACL, действительный в течении этой сессии, попутно произведя его оптимизацию путем суммирования прав различных *групп доступа* к одним и тем же *объектам доступа*. С точки зрения совершенствования аудита, это обеспечит точную фиксацию: какой субъект работал с системой в режиме чтения, когда и с какими данными, а для разграничения доступа – позволит использовать более сложные процедурные алгоритмы проверки полномочий, выполняемые на стадии регистрации пользователя в системе (например – появится возможность реализовать дополнительную авторизацию).

2.3.9.4. SAMCapability – привилегии

Таб. 12 SAMUserCapability (XUC) – Привилегии пользователя

#	Поле	Тип	Название	Описание
01	**uid**	bigint	Пользователь	ID пользователя системы
02	**sid**	bigint	Область данных	ID области данных, 0 – для глобальных привилегий
03	**capcode**	char(32)	Привилегия	код привилегии, разрешенной данному пользователю

Разграничение доступа к некоторым операциям не укладывается в концепцию ACL. Причины этого могут быть различны – некоторые действия не связаны с конкретными объектами доступа, для других – использование ACL слишком громоздко, третьи являются некоторым особым способом совершения стандартной операции, возможность использования которого следует контролировать отдельно. Для контроля доступа к таким функциям системы предназначен механизм *привилегий* (*capability*).

Привилегия (*capability*) – это право субъекта на использовании некоторой функции системы. В TIS/SQL привилегии назначаются каждому пользователю индивидуально, через таблицу SAMUserCapability (Таб. 12), и бывают двух типов:

1. *Привилегия на область данных* – разрешает использовании защищаемой функции системы в пределах указанной области данных.
2. *Глобальная привилегия* – разрешает пользоваться этой функцией системы во всех областях данных. Используется, также, для операций, при которых понятие *области данных* неприменимо в качестве ограничителя.

Базовый набор привилегий TIS – приведен в Таб. 13. Список привилегий, существующих в системе, проще всего хранить в *стандартном словаре*, зарезервированный код словаря – "CAPABILITIES". Стандартизовать механизм того, как и где привилегия ограничивает доступ к конкретной функциональности системы – невозможно, поэтому TIS/SQL стандартизует только интерфейс работы с привилегиями (назначить/убрать/проверить), а остальное отдается на откуп конкретной реализации, конкретной функциональности.

Назначение пользователю привилегий – не делает его *суперпользователем*. Привилегии – это просто еще один стандартный механизм разграничения доступа, имеющий свою область применения. Термин capability (буквально - «способность») – гораздо лучше передает его рутинность и ординарность, однако пока не имеет устоявшегося русскоязычного эквивалента.

Таб. 13 Коды привилегий

Код	Название	Описание
SAM_MANAGE	Управление СРД	Глобальная привилегия - позволяет редактировать любые данные подсистемы SAM.
SAM_MANAGE_USERS	Управление пользователями	Привилегия на область данных – позволяет создавать и редактировать в ней пользователей, а также управлять их участием в заранее подготовленные группах доступа, принадлежащих той же области данных.
SAM_MANAGE_ACL	Редактирование ACL	На область. Редактирование ACL групп, принадлежащих этой области данных, по отношению к данным в этой же области данных.
SLVL_DECREASE	Понижать уровень секретности	Глобальная или на область. При изменении уровня секретности ОД, позволяет выполнить его понижение.
SLVL_MANAGE	Изменять уровень секретности	Глобальная или на область. Позволяет изменять уровень секретности существующих ОД. (в общем случае, изменение возможно только в сторону повышения)
SLVL_MANAGE_HISTORY	Изменять уровень секретности истории изменений ОД	Глобальная или на область. Позволяет изменять уровень секретности существующих ОД, с одновременным установлением такого же уровня секретности для всех предыдущих его версий. В более сложных реализациях – изменять уровень секретности каждой версии ОД индивидуально.
SLVL_REQUEST	Запрашивать изменение уровня секретности	Глобальная или на область. Позволяет помещать ОД в транзит, с запросом на изменение уровня секретности. Окончательное решение принимает пользователь с привилегией SLVL_MANAGE
TRUSTED_USER	Доверенный пользователь	Глобальная – пользователь имеет право назначить своей сессии с БД любого пользователя системы. На область – то же, но только пользователей определенных в этой области.
MOVE_WITH_HISTORY	Перемещение истории ОД	Глобальная – Перемещение ОД между областями, вместе с историей изменений.
MOVE_SEND	Отправка ОД	На область – отправка ОД из этой области.
MOVE_RECEIVE	Прием ОД	На область – прием (или отказе в приеме) ОД отправленных в эту область, а также их просмотр, во время нахождения в транзите.
FORCE_ROLLBACK	Откат чужих транзакций	Глобальная – отмена логических транзакций, начатых другими пользователями.
REPLICA_DO_CHPTS	Выполнение CHPTS	На область – установка значения `SAMScope.chpts` в текущие дату/время. Т.е. любая незавершенная логическая транзакция в этой области, должна получить дату/время завершения после chpts
REPLICA_MAKE	Подготовка пакета репликации	На область – подготовка пакета изменений, произошедших в области данных, за период времени.
REPLICA_LOAD	Загрузка пакета репликации	На область – загрузка пакета изменений в области данных.
AS_SQL_QUERY	Выполнение свободных SQL запросов	Глобальная – проверяется сервером приложений, имеющим функциональность исполнения свободных SQL запросов.

Таб. 14 Категории привилегий (префиксы кодов привилегий)

Префикс	Категория привилегий
SAM	Управление данными подсистемы SAM
SLVL	Управление метками мандатной СРД
MOVE	Перемещение объектов данных между областями данных
REPLICA	Репликация содержимого области данных
AS	Application Server. Привилегия контролируется сервером приложений

2.3.9.5. SLevel – уровень секретности, мандатная СРД

Последний, третий механизм разграничения доступа, предусматриваемый TIS подходом, это *мандатная система разграничения доступа*. Именно с наличием такого механизма, обычно, ассоциируется характеристика «*trusted*» в названии программного продукта. Существует множество взаимодополняющих «моделей» (наборов правил) для построения мандатных СРД, дальнейшее изложение ориентируется на правила, диктуемые российскими государственными органами.

Суть мандатного механизма состоит в сопоставлении *метки конфиденциальности*, присущей *объекту доступа*, и *уровня доступа*, заданного *субъекту доступа;* сравниваемые метка и уровень должны принадлежать к одной *категории*. Допускается одновременное существование нескольких таких *категорий*. Отсутствие доступа в рамках любой категории – означает отказ в доступе, даже если он разрешен в рамках любых других категорий или через ACL, а разрешение доступа в пределах всех категорий – не означает, что пользователь этот доступ получит, а всего лишь позволяет его получить, если доступ разрешен одним из дискреционных методов (например – ACL). Все это дополняется еще одним ключевым ограничением – в рамках мандатного механизма, *субъект доступа*, не может делегировать свои права доступа другому *субъекту*, назначение *уровней доступа* – прерогатива администратора безопасности.

Категории бывают двух типов – *иерархические* и *неиерархические*. Суть неиерархических категорий состоит в том, что для получения доступа к объекту, в метке которого проставлены некоторые категории, субъект должен обладать всеми этими категориями. Наглядный пример из повседневной жизни – категории транспортных средств (ABCDE) и права на управление ими. В TIS/SQL неиерархические категории не используются. Впрочем, идеологического запрета на этот способ разграничения доступа нет (равно как и на применение других «моделей»), просто концепция *областей данных* способна заменить их в наиболее распространенных случаях. Понятие *области данных* — изначально родственно концепции *неиерархических категорий*, а механизм *ACL на область данных* – обладает многими чертами мандатной СРД, но при этом используется в качестве дискреционного метода.

Иерархические категории подразумевают численное выражение уровней доступа и *меток конфиденциальности* таким образом, что при их сравнении возможны три результата – больше, меньше или равно. Концепция иерархических категорий развилась из грифов секретности документов, используемых в государственной и военной бюрократии. В наших условиях, с некоторыми дополнениями, это (в порядке возрастания секретности): не секретно (НС), для служебного пользования (ДСП), секретно (С), совершенно секретно (СС), совершенно секретно особой важности (ССОВ). Впредь условимся, что более высокому уровню, соответствует большее численное значение, тогда принцип работы иерархических категорий следующий (Bell-LaPodula model):

- субъект может *читать* объект, если его уровень доступа *больше* или равен уровню объекта.
- субъект может *писать* в объект, если его уровень доступа *меньше* или равен уровню объекта.

Если первый пункт очевиден, то второй – зачастую вызывает отторжение, непонимание и подозрения на опечатку в формулировке. Все дело в том, что корректно работающая мандатная СРД должна не только защищать информацию от прямого несанкционированного доступа, но и препятствовать несанкционированному понижению ее уровня секретности. Возможность одновременно читать документ, и писать другой, менее защищенный – есть неконтролируемый канал утечки информации.

Таким образом, пользователь может читать и писать только те документы, *уровень секретности* которых совпадает с его *уровнем доступа;* документы с меньшим уровнем секретности – только читать; документы с большим – только писать. На практике, для нормальной работы одного пользователя с документами различных грифов секретности можно пойти тремя путями:

1. Субъект всегда создает объект с максимально доступным ему уровнем секретности, после чего объект проходит процедуру рассекречивания. Формально – самый корректный способ работы.
2. Пользователю предоставляется несколько учетных записей, с различными уровнями секретности, и он по мере необходимости работает под той учетной записью, которая позволяет редактировать нужные документы. В пределах одной сессии – все правила работы соблюдаются.
3. При инициализации сессии с БД, пользователь задает максимальный уровень доступа, которым он будет пользоваться, и это ограничение действует до завершения сессии. Данный способ – разновидность предыдущего, но требует со стороны СУБД – расширенной поддержки идентификации

сессий, а со стороны разработчиков системы – дополнительной проработки и согласования деталей реализации. (также см «Идея динамических таблиц СРД» в разделе посвященном ACL)

Следующая сложность – данные не имеющие метки мандатной СРД и правила обращения с ними. Формально, таковых в системе быть не должно, и для данных прикладных подсистем это действительно так – любая запись имеет стандартный заголовок, в котором есть специальное поле ZLVL. Однако служебные таблицы, принадлежащие словарю или подсистеме SAM, не имеют таких полей-меток, а их введение излишне усложнит конструкцию системы. При этом следует осознавать, что переписывание конфиденциальной информации в описательные поля служебных данных – является потенциальным каналом утечки информации. Решением является признание всех служебных таблиц хранилищем несекретной информации, с соответствующими ограничениями на редактирование. В качестве идеи, можно рассмотреть присвоение служебным данным априорного уровня секретности ниже чем «не секретно», что защитит их от редактирования любым ординарным пользователем.

В том что администратор безопасности, имея уровень доступа «не секретно» или ниже, управляет учетными записями, задавая им любой уровень доступа – просматривается некоторая идеологическая коллизия. Такие же коллизии присутствуют при доступе в систему администратора БД, создании и восстановлении резервных копий БД, или при создании и загрузки репликационных пакетов изменений в сложных «многоядерных» системах. Изящного, универсального решения они не имеют – это всегда компромисс из технических средств и организационных мероприятий, достигнутый между разработчиком, потребителями, сертифицирующими и контролирующими органами.

Реализация мандатной СРД в TIS/SQL состоит всего из одной иерархической категории SLevel, определяющей гриф секретности объектов и уровень доступа субъектов. Метка конфиденциальности хранится в специальном поле ZLVL стандартного заголовка, а уровень доступа субъекта – задается в поле SAMUser.slevel. Следует отметить, что уровень доступа не является меткой конфиденциальности, описание пользователя всегда имеет гриф «не секретно». Примерный набор уровней секретности представлен в Таб. 15; рекомендуется назначать численные значения с некоторой скважностью, дающей пространство для маневра. Также рекомендуется предусмотреть контроль качества для значений ZLVL, что-бы пользователи не порождали документов с уровнями секретности, для которых еще нет определения.

При реализации мандатной СРД следует определиться с тем, что подразумевать под записью ОД, имеющих уровень секретности больший, чем уровень доступа пользователя. Очевидно, что свободное редактирование пользователем с доступом «С», документов с грифом «СС» – не то, что требуется от СРД. За отправную точку можно принять запрет на создание и редактирование документов, с грифом секретности выше, чем максимально допустимый уровень доступа пользователя.

Возможность редактирования меток конфиденциальности проще всего реализовать через механизм привилегий. В Таб. 13 представлен их комплект, достаточный для первоначальной реализации. Особо следует ограничивать и контролировать возможность понижения уровня секретности.

Обычно, изменение метки конфиденциальности должно происходить как создание новой версии ОД, таким образом, что все записи, принадлежащие текущей версии – помечаются как утратившие актуальность, а вместо них создаются копии, с новым номером версии ОД и новым значением ZLVL. В общем случае, все записи ОД в пределах одной версии имеют один тот же ZLVL, а в каждой записи он продублирован только для ускорения операций контроля доступа.

Еще одна проблема, с реализацией мандатной СРД в TIS подходе, является прямым следствием хранения полной истории всех изменений. Следует всегда помнить о том, что помимо последнего состояния ОД, существуют предыдущие, зачастую содержащие ту же самую информацию. Если мы меняем уровень конфиденциальности документа, особенно если речь идет о его повышении, следует задуматься и об изменении уровня конфиденциальности его предыдущих версий.

Таб. 15 Уровни секретности ZLVL, slevel

ZLVL	Гриф	Расшифровка аббревиатуры грифа секретности
0	НС	Не секретно
10	ДСП	Для служебного пользования
20	С	Секретно
30	СС	Совершенно секретно
40	ССОВ	Совершенно секретно особой важности

2.3.9.6. SAMAudit – журнал аудита

Таб. 16 SAMAuditEvent (XE) – Журнал аудита

#	Поле	Тип	Название	Описание
01	**id**	bigint	ID записи	Уникальный идентификатор записи или события
02	uid	bigint	Пользователь	ID пользователя, совершившего действие
03	sid	bigint	Область данных	Область данных, в которой совершено действие
04	obj_type	char(5)	Тип объекта	Код типа ОД, записи или поля, которого касается это событие
05	ZID	bigint	ID объекта	ZOID объекта, ZRID или другой id записи
06	ts	timestamp	Время события	Дата/время события
07	capcode	char(32)	Привилегия	Код использованной привилегии
08	eventcode	smallint	Код события	Числовой код события
09	eventinfo	varchar(255)	Описание	Свободное описание события, с указанием его специфических параметров

Последняя часть СРД, придающая ей полноценность и завершенность – это *аудит* всех действий *субъекта* в системе. Под аудитом понимают независимый контроль за функционированием системы вообще, и выполнением правил и норм по работе с информацией в ней содержащейся, в частности. Для того, чтобы такой контроль был возможен, система должна накапливать информацию обо всех действиях в ней происходящих, что осуществляется с применением *журнала аудита*, в который записывают все происходящие события, с параметрической и описательной информацией, необходимой для последующего анализа.

Сложность реализации *журнала аудита* состоит в том, что процесс сбора информации должен быть встроен во все части системы, т.е. получается что он «размазан» по всему ее коду, что с одной стороны достаточно трудоемко, а с другой – сложно поддается тестированию, которое способно подтвердить, что все работает правильно, и нигде ничего не упущено. Отдельную сложность представляет разработка структуры данных для журнала, поскольку это всегда подразумевает некоторое упрощение реальной картины. И наконец, *журнал аудита*, это часть системы, требующая больших затрат на реализацию и поддержание, но при этом не обслуживающая никаких прикладных потребностей, ради которых разрабатывается ИС. Все вышесказанное приводит к тому, что в большинстве реально эксплуатируемых систем, реализация *журнала аудита* очень далека от идеала.

В TIS/SQL подходе для целей аудита может и должна использоваться накопленная *история изменений*, но полностью заменить ведения *журнала аудита* она не может – всегда найдутся действия, которые не отражаются в истории изменений, но важны для аудитора.

Вообще, стандарты на обработку информации, требуют чтобы в журнале аудита фиксировались все успешные и неуспешные попытки получить доступ к *объектам доступа,* как в режиме записи, так и в режиме чтения. Если с записью все более-менее просто, поскольку она осуществляется только через вызовы хранимых процедур, то с аудитом доступа в режиме чтения у TIS/SQL подхода возникают существенные сложности. Поскольку чтение осуществляется обычными SQL запросами к фильтрующим view, то универсального способа учета, к какой информации пользователь действительно получил доступ – нет. В свете этой проблемы следует рассмотреть следующие варианты:

1. Считать, что пользователь получил доступ ко всей информации, которая могла быть доступна ему в течении сессии работы с системой. Иными словами – не делать ничего.
2. Перейти на чтение информации только через вызовы хранимых процедур. Это наиболее универсальный метод, но система утрачивает большинство достоинств реляционной БД. При этом фильтрующие view – следует сохранить, запретив доступ к ним ординарным пользователям, а процедурам чтения – получать запрошенные данные только через эти view, что позволит по крайней мере внутри *ядра системы* пользоваться всей мощью реляционного подхода.
3. Использовать в фильтрующих view данных из динамических таблиц СРД, в которых должны быть перечислены те *области данных* и/или *объекты данных*, которые пользователь намерен читать. Заполнение этих динамических таблиц осуществляется вызовами хранимых процедур, что позволяет записывать, какие данные и в какой период времени пользователь мог прочитать.
4. Использовать встроенные средства СУБД для аудита SQL запросов выполненных пользователем.

5. Перенести часть функций аудита в сервер приложений.
6. Осуществить гибридизацию перечисленных вариантов и получить новый, удовлетворяющий техническому заданию на информационную систему.

Еще одна существенная проблема с ведением журнала аудита внутри БД – это проблема транзакций и контроля над их завершением. Суть в следующем:

- Субъект начинает физическую транзакцию и выполняет вызов к хранимой процедуре
- Хранимая процедура выполняет запрошенные действия и делает запись в *журнал аудита*
- Пользователь просматривает результат выполнения, после чего выполняет откат начатой им транзакции.
- Если журнал ведется в БД – событие оказалось не зафиксированным, если за ее пределами – событие зафиксировано, но действие на самом деле не выполнено.

В большинстве существующих СУБД, возможность хранимых процедур контролировать начало и завершение транзакций – не является абсолютной. До некоторой степени с этим можно мириться, но при построении систем высоких классов защищенности – БД следует изолировать от непосредственного доступа пользователей. Осуществить это можно с помощью специального *сервера приложений*, который должен будет принимать запросы пользователей, транслировать их в обращения к ядру системы, от имени этого пользователя, обеспечивая при этом необходимые характеристики среды исполнения, после чего возвращать результат пользователю-инициатору запроса. В таких системах TIS/SQL подход обеспечивает, при соблюдении некоторых граничных условий, правильное разграничение доступа, аудит, ведение истории и целостность данных, а промежуточный сервер приложений – гарантирует выполнение этих условий.

Структура данных для журнала аудита, представленная на Рис. 4 и в Таб. 16 – примитивна и схематична, но ее будет достаточно для прототипа системы. При переходе к следующему этапу разработки она должна быть подвергнута критическому переосмыслению. Впрочем, многие реальные системы на самом деле не нуждаются в безупречном аудите, для них он является скорее бесплатным приложением к тематическим функциям, которое включается только из любопытства или для целей отладки. Предложенный журнал аудита подразумевает несколько уровней «глубины» учитываемых событий:

1. события учитываемые всегда;
2. события, важные для аудита, но учет которых можно включить или отключить;
3. события, важные только для с точки зрения отладки, учет которых можно включить/отключить.

Включение и отключение разных уровней аудита осуществляется для каждой области данных и каждого типа ОД индивидуально, в таблице SAMScopePolicy (Таб. 7). В десятичном значении кода события (SAMAuditEvent.eventcode) можно закодировать характеристики фиксируемого события, например следующим образом – LNNPPRR, где

- L – уровень (класс) события (0 – системное неотключаемое, 1 – доступ к данным неотключаемое, 2 – доступ к данным отключаемое, 3 – предупреждение отключаемое, 4 – отладочное)
- NN – тип запроса пользователя или произошедшего произошедшего события (01 – чтение, 02 – запись, 03 – создание, 04 – удаление, 05 – блокировка, 06 – снятие блокировки, ..., 91 – повышение ZLVL, 92 – понижение ZLVL и т.д.)
- PP – точка алгоритма по обработке события, в котором осуществляется аудит, в предположении, что обработка одного запроса может потребовать нескольких записей в журнале (00 – главная точка контроля, 01 – точка входа в процедуру, 99 – точка выхода из процедуры).
- RR – код причины отказа (00 – выполнено успешно)

Например код 2030000 – будет означать успешно произошедшее событие «создание» объекта типа obj_type, в области sid, с присвоением ему идентификатора ZID, при этом событие относится к классу отключаемых.

Используемые коды для {L,NN,PP,RR} – следует, снабдив расшифровкой, поместить в *словарь*.

Некоторые пояснения следует сделать относительно поля `id` – оно обозначено как первичный ключ, но, на самом деле, в этом качестве его польза весьма сомнительна, да и почти все современные СУБД позволяют создавать таблицы без первичного ключа. Гораздо интереснее использовать его в качестве общего идентификатора для всех записей, относящихся к одному событию.

2.3.10. Dictionary – словарь, система символьных кодов

Таб. 17 ZCode (ZC) – Основной словарь (V_ZDic)

№	Поле	Тип	Описание
01	**dic**	char(16)	код (идентификатор) словаря
02	**code**	char(32)	код (напр коды стан – RU,US,UK; валют – RUR, USD, GBP; и т.п.)
03	value	varchar(99)	значение (краткая расшифровка, используемая вместо кода при отображении документа)
04	descr	varchar(255)	описание (расширенное описание кода и его применения)

Словарь (*dictionary*) – это совокупность таблиц, в которых задается некоторая настроечная информация, влияющая на поведение системы, в частности – на процесс *контроля качества*. Содержимое словаря, будучи обычными записями в таблицах, скорее должно рассматриваться как конструктивные элементы ИС, наравне со структурой БД, чем как пользовательские данные.

В простейшем случае, словарь состоит из одной единственной таблицы ZCode, каждая запись которой является описанием код в некотором словаре. View V_ZDic отображает эту таблицу без какой-либо фильтрации. Разница наименований таблицы и view обусловлена желанием установить некоторый идеологический посыл для разработчиков «*библиотеки доступа к данным в терминах объектов*» (см Рис. 1): ZCode – структура данных для хранения одной записи словаря, а ZDic – это структура для хранения всего множества кодов одного словаря, причем не обязательно из таблицы ZCode. Наиболее распространенный способ применения словарей – это ограничение возможных значений поля, кодами из конкретного словаря. При этом, если данное множество значений используется только для этого поля, то код словаря совпадает с кодом поля.

Символьные коды ограниченной длины широко используются в TIS/SQL подходе для «место-именного» обозначения конструктивных частей системы – типов ОД, таблиц и полей. Все эти коды должны присутствовать в специальных словарях `'TIS_OBJECT_TYPE'`, `'TIS_TABLE'`, `'TIS_FIELD'`, `'TIS_FIELD_DEF'` (словарь `'TIS_FIELD'` удобно использовать в прикладном ПО, чтобы централизованно контролировать отображаемые им имена полей). Например, для приводимых в следующем разделе типа ОД «`Document`» и таблицы «`DRef`», входящей в его структуру (см. Рис. 5 и Таб. 8), заполнение этих словарей скриптом может выглядеть так:

```
select set_code('TIS_OBJECT_TYPE','D','Document','Пример документарного ОД');
select set_code('TIS_TABLE','DR','DRef','Ссылка на документ');
select set_code('TIS_FIELD','DR01','Документ','Ссылка на ОД типа "документ"');
select set_code('TIS_FIELD_DEF','DR01','doc_id','bigint;REF;UNIQ=OBJ');
select set_code('TIS_FIELD','DR02','Тип отношений', 'использует/заменяет/дополняет...');
select set_code('TIS_FIELD_DEF','DR02','reltype','char(2);DIC');
select set_code('DR02','U','использует','документ-ссылка использован при подготовке');
select set_code('DR02','R','заменяет','документ-ссылка заменен этим документом');
select set_code('DR02','A','дополняет','этот документ дополняет документ-ссылку');
```

Словарь ZDic покрывает большую часть потребностей системы, но иногда возникают документы, для описания корректной структуры которых необходимо дополнить словарь новыми таблицами, зачастую – весьма замысловатой структуры.

Все таблицы словаря выполняются в рамках классической реляционной модели, чтобы не затенять смысла этих конструкций. Изменение данных словаря – прерогатива администратора БД. Для ввода информации в словарь, обычно создается одна-две простейших хранимых процедур на таблицу, а все данные словаря хранятся и поддерживаются в виде скрипта, состоящего из вызовов этих процедур, который можно в любой момент загрузить в БД, что приведет к удалению всех существующих данных словаря и загрузки нового состояния. Возможен полный или частичный уход от этого способа в сторону реализации словаря с применением концепции *объектов данных*, когда его «классические» реляционные таблицы превратятся в *индексные таблицы* к ОД подсистемы «Словарь», но накладные расходы на разработку и поддержание такой реализации мало-оправданы для большинства ИС.

Поскольку словарь это скорее часть структуры данных, чем сами данные – в репликативных системах данные словаря между инсталляциями не передаются, а поступают из некоторого центра, в виде скрипта, который системный администратор должен будет загружать вручную.

2.3.11. Квази-ER диаграмма, описания таблиц

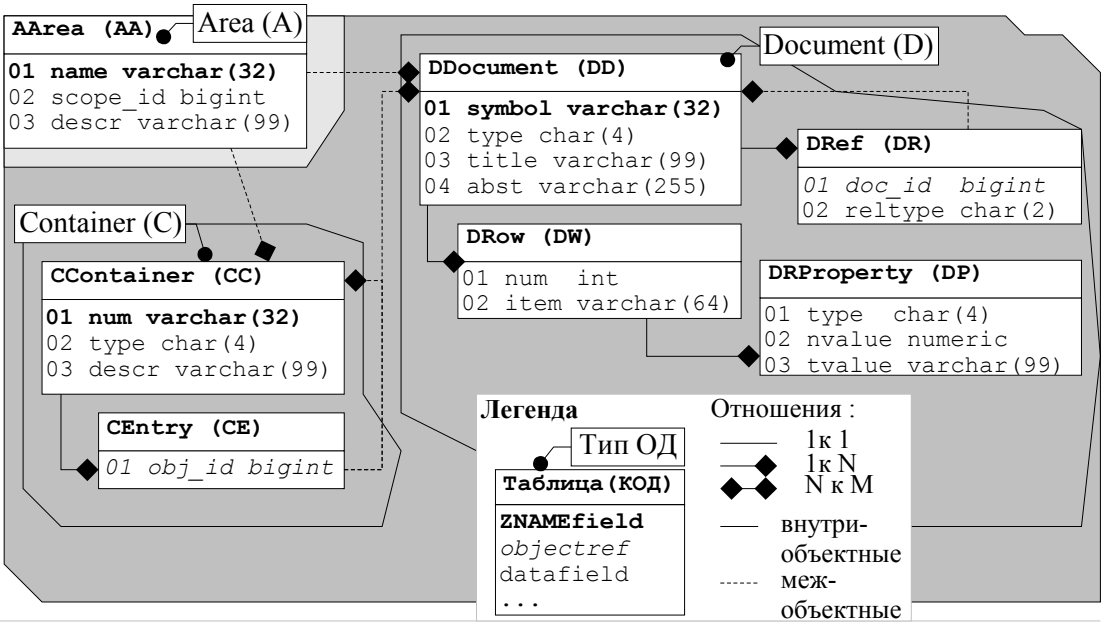


Рис. 5 Квази-ER диаграмма (пример)

При проектировании структур реляционных БД обычно используется такое понятие как Entity/Relation диаграмма (ER диаграмма). Первоначальная область применения этого понятия гораздо шире, но сейчас первая ассоциация со термином ER-диаграмма - «картинка, на которой изображена структура таблиц реляционной БД и обозначены все связи между ними». Подобная ER-диаграмма недостаточно внятно отображает структуру TIS/SQL БД, чрезмерно затеняя предметную область несущественными деталями. Что-бы подчеркнуть особенности ER диаграмм, используемых TIS/SQL подходом, для них используется термин «*квази-ER*». Вот список особенностей таких диаграмм:

- Таблицы отображаются без полей стандартного заголовка.
- Обозначены *типы объектов данных*, с указанием их границ (таблиц входящих в структуру ОД).
- Обозначаются только наиболее важные связи внутри ОД и между ними (без учета «истории»).
- Все типы ОД, таблицы и поля снабжены кодами. (для полей полный код это: код_таблицы + "номер_поля", указанный на диаграмме)
- Для доступа к данным программисты должны использовать не сами таблицы, а view, имена которых порождаются от имен таблиц по отдельно оговоренным правилам (напр. префикс V_).
- На квази-ER диаграмме не отображаются индексные таблицы, таблицы SAM, словаря и системные таблицы (ZObject и т.п.).

Квази-ER диаграмма – это основной визуальный документ, который постоянно лежит на столе у разработчиков, но он недостаточно полон и строг для описания структур таблиц, связей, характеристик полей и т.п. TIS/SQL подход требует наличия такого описания, которое можно использовать в качестве исходных данных для кодогенератора, поэтому для каждой таблицы должно быть подготовлено описание, подобное представленному в Таб. 18. Представленный вариант «языка описания таблиц» – не догма, просто он достаточно компактен, нагляден и, одновременно, легко поддается обработке скриптами на awk или perl, если таблицу преобразовать в текст с разделителями полей.

Таб. 18 DRef (DR) – Ссылка на документ (пример описания таблицы)

# Атрибут			**Значение**	**Значение2/код**	
NAME			DRef	DR	
PARENT			DDocument	DD	
OBJECT			Document	D	
HEADER			STD_HEADER		
INDEX			doc_id		
#	**Поле**	**Тип**	**Атрибуты**	**Название**	**Описание**
01	doc_id	bigint	REF;UNIQ=OBJ	Документ	Ссылка на ОД типа «документ»
02	reltype	char(2)	DIC	Тип отношений	использует/заменяет/дополняет документ doc_id

2.4. Распределенные и репликативные системы

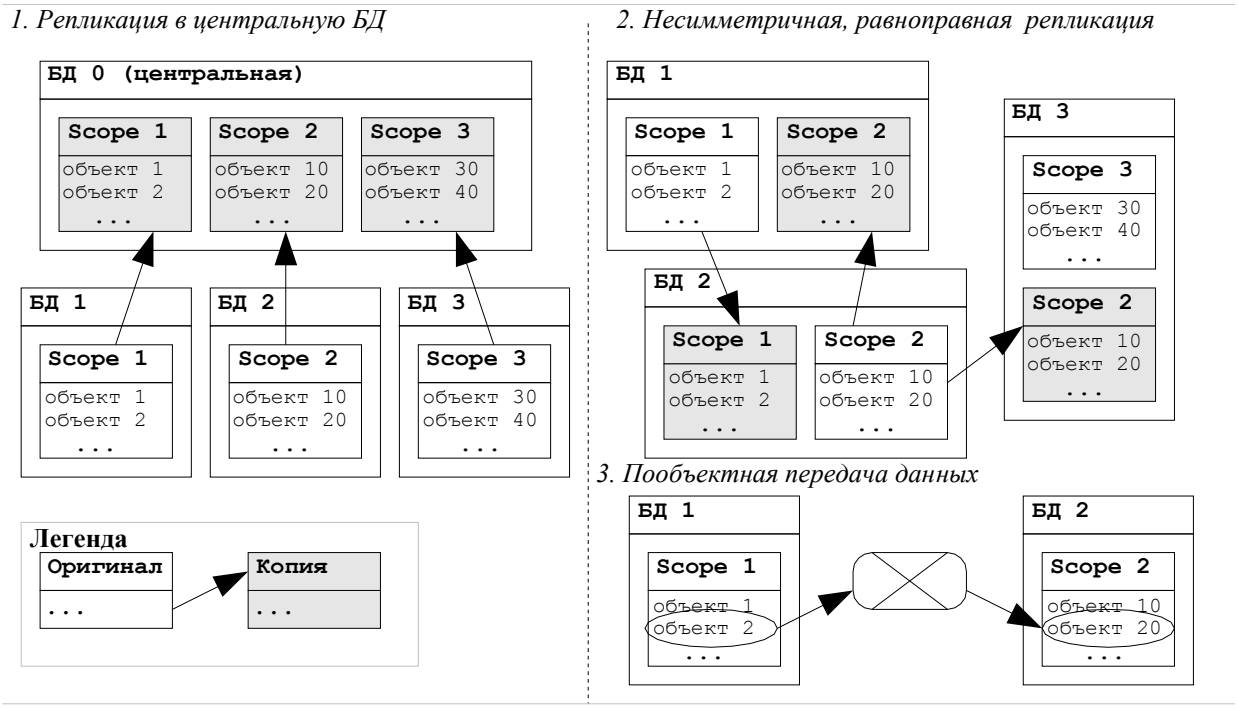


Рис. 6 Варианты схем репликации

TIS/SQL подход позволяет совместить в пределах одной БД практически любое количество подсистеме, абсолютно произвольного назначения. Хранение всей информации в единой БД придает ей некоторое новое качество, открывая широкие возможности для анализа и контроля над происходящим в организации в целом. Сопровождение (как эксплуатация, так и разработка) одной информационной системы, обслуживающей несколько задач, обходится значительно дешевле, чем нескольких независимых ИС для каждой задачи. Таким образом, базовой и рекомендуемой схемой для ИС, базирующейся на TIS/SQL подходе, следует считать систему с одной централизованной БД для всех обслуживаемых задач.

На практике, существуют случаи, когда одной БД бывает недостаточно, и необходимо иметь несколько инсталляций системы, осуществляя обмен информацией между ними. Вот список наиболее распространенных подобных ситуаций:

1. Географически распределенная организация, с большим объемом транзакций в филиалах.
2. Интеграция систем, разработанных независимо.
3. Особые требования к разграничению доступа, требующие физического разделения систем. (Например – обмен данными между независимыми организациями.)
4. Требования к производительности отдельных подсистем .
5. Балансировка нагрузки на систему.

Во всех этих случаях требуется строить распределенную систему, состоящую из нескольких, слабосвязанных между собой, инсталляций, регулярно (или нерегулярно) обменивающихся информацией в виде репликационных пакетов. Проблемы, возникающие при построении таких систем, равно как и способы их решений – тема необъятная, по объему выходящая не только за допустимые границы этой главы, но всей работы в целом, поэтому здесь пойдет речь только об основных свойствах TIS/SQL, которые могут быть задействованы для решения подобных задач. Все эти свойства уже описаны ранее, а сейчас на них необходимо взглянуть через призму «распределенно-репликативной ИС».

Во первых – *область данных.* В это понятие изначально заложена изолированность находящихся в ней данных, достаточная для их переноса в отдельную инсталляцию системы. В этом случае, только одна инсталляция системы должна содержать «главную» копию *области данных*, т.е. ту в которую допускается внесение изменений пользователями, во всех остальных – может находиться

только ее статичная копия, представляющая собой срез состояния этой области на определенный момент времени (Рис. 6). Поскольку все данные имеют метку принадлежности области данных, не составит особого труда отделить изменения, произошедшие в конкретной области, за некоторый период времени. Далее, этот комплект изменений можно передать другим инсталляциям системы, в которых находятся статичные копии, чтобы привести их состояние к новому моменту времени. В таблице SAMScope (Таб. 6) предусмотрены специальные поля: is_local и chpts, обслуживающие базовые потребности распределенных систем.

Во вторых – *объект данных*. ОД – достаточно изолированная сущность, чтобы можно было выгрузить его из одной БД и, на основе этой информации, – создать копию в другой (Рис. 6 схема 3). При «по-объектной передаче» в один пакет можно поместить более одного ОД, что позволит при загрузке сохранить связи между ними. Эта схема подходит для абсолютно независимых инсталляций, и является скорее основой для систем электронного документооборота, чем для распределенных БД.

Впрочем, одно другому не мешает – если система подразумевает активное движение ОД между *областями данных*, расположенными в разных БД, то наиболее безопасный, с точки зрения логической целостности БД, механизм их транзита – начинать существование в новой системе «с чистого листа», дабы шлейф предыдущей истории не мог наложиться на это новое состояние ОД, в неожиданном месте, с неожиданными последствиями. Например:

1. из БД1 в БД2 отправлен ОД1
2. не дожидаясь подтверждения о его приеме – ОД1 отозван в БД1 и отредактирован
3. в то же время, в БД2, ОД1 успешно принят
4. при репликации БД1 и БД2 в БД3 обнаружиться, что ОД1 существует в двух экземплярах.
5. разрешение этой коллизии гораздо проще, если это два разных ОД.

Вообще, проблема *транзита*, в распределенных системах, - требует тщательной проработки.

Еще несколько проблем, возникающих при построении распределенной системы, заслуживают упоминания здесь, и пристального внимания со стороны архитектора и разработчиков ИС. Первая проблема – это поддержание уникальности идентификаторов ZOID во всех инсталляциях. Здесь возможные различные решения:

- Радикальное решение – уникальность ZOID только в пределах области данных. В этом случае придется пересмотреть многие алгоритмы, поэтому такое решение лучше принимать на начальном этапе разработки.
- При построении системы, близкой к схеме 1 (Рис. 6) «с нуля» - возможно выделение для каждой инсталляции системы своего диапазона идентификаторов. В этом случае возникают дополнительные организационные затраты на ведение реестра диапазонов.
- При интеграции независимых систем – можно разработать согласованный механизм трансляции идентификаторов одной системы в идентификаторы другой. Для «по-объектного обмена» это единственный вариант решения. При трансляции идентификаторов необходимо не забыть обеспечить правильную трансляцию всех ссылок.

Вторая проблема: последнее актуальное состояние БД, содержащей копии «чужих» областей данных, – это всегда смешение разных временных срезов, т.е. состояние, которое никогда не существовало в реальности. Об этом следует помнить при порождении отчетных документов, которые в этой ситуации следует получать на основе исторических данных, по временному срезу не позднее, чем самый меньший SAMScope.chpts.

Третья проблема – это обеспечение единого времени во всех инсталляциях системы. Здесь целесообразно рассмотреть переход от локального времени к UTC (Универсальному Координированному).

И самая главная, четвертая проблема – загружая в свою БД репликационный пакет из неконтролируемого вами источника, вы рискуете получить информацию не соответствующую локальным критериям качества. Система должна быть устойчива к ошибкам в данных, привнесенным таким путем. Вообще – любой репликационный пакет может служить средством атаки на вашу систему, это следует понимать, всегда об этом помнить и быть готовым к решению подобных проблем.

2.5. Хранение и воспроизводимость истории. Подготовка отчетных документов.

Как уже неоднократно говорилось, одним из фундаментальных свойств TIS/SQL подхода является хранение истории всех изменений таким образом, что возможно воспроизвести состояние прикладных данных на любой момент времени в прошлом. Способ хранения «исторический» информации таков, что доступ к ней возможен в любое время, и не требует каких-либо специальных подготовительных действий со стороны администратора системы.

Доступность «исторических» данных для ординарных пользователей системы рождает ряд проблем для СРД и прикладного ПО:

- При разграничении доступа необходимо отдельно проработать вопросы доступа к историческим данным, о чем уже рассказывалось ранее, в разделах посвященных ACL и мандатной СРД.
- Для прикладного ПО проблема состоит в том, что любая ошибка в данных, даже после ее исправления, продолжает существовать в БД, и будет воспроизведена при просмотре ее среза на соответствующий момент времени. Из этого следует, что при разработке любого ПО для TIS/SQL необходимо предусматривать его правильное поведение на «некорректных» прикладных данных, поскольку всегда есть вероятность, что ему придется работать с некоторым срезом БД, во время существования которого отсутствовали правила и ограничения действующие в настоящий момент.

Еще одна неочевидная, и поэтому каверзная, проблема возникает когда появляется потребность просмотреть некоторую предшествующую версию ОД. Поскольку редкий документ в ИС существует без связей с другими, то воспроизведение версии только одного из них, вкупе с действующими версиями «зависимостей», – дает недостоверную картину, никогда не существовавшую в действительности. В некоторых случаях с этим можно мирится, в некоторых – нет. Для получения достаточно достоверной картины — необходимо построить срез состояния всей БД на заданный момент времени. При этом следует помнить о том, что в системе есть данные (словарь, SAM, индексные таблицы, системные таблицы СУБД), для которых состояние, в общем случае, не воспроизводится.

Что касается техники работы с историческими данными, то она базируется на фильтрующих view, которые должны уметь отображать все исторические данные, или конкретный срез БД, на момент времени заданный пользователем. В общем случае, для каждой таблицы `SomeTable` есть три view:

- `V_SomeTable` – последнее актуальное состояние
- `VH_SomeTable` – все версии, всех записей,для которых пользователю разрешено чтение истории.
- `VD_SomeTable` – состояние на предварительно заданный пользователем момент времени.

Таким образом, для работы с любым срезом времени, необходимо задать желаемый момент времени, после чего система view VD_* должна будет отображать соответствующее ему состояние БД. Именно эту систему view рекомендуется использовать для построения отчетных документов, что-бы обеспечить их воспроизводимость.

Начиная строить отчетный документ, следует позаботится о неизменности данных в течении всего всего времени выполнения программы, его готовящей. Это можно обеспечить различными способами, предоставляемыми СУБД. TIS/SQL подход добавляет к ним еще один – зафиксировать момент текущий момент времени через установку `SAMScope.chpts` (см. Таб. 6) всех используемых областей, установить этот зафиксированный момент в качестве среза БД, доступного через VD_*, и получать отчетный документ по этому срезу данных.

> **NB** В реплицируемых системных, достоверные отчетные документы можно получать только на дату не позднее самого раннего chpts нелокальных областей.

3. Правила организации, хранения и обработки данных ядром системы

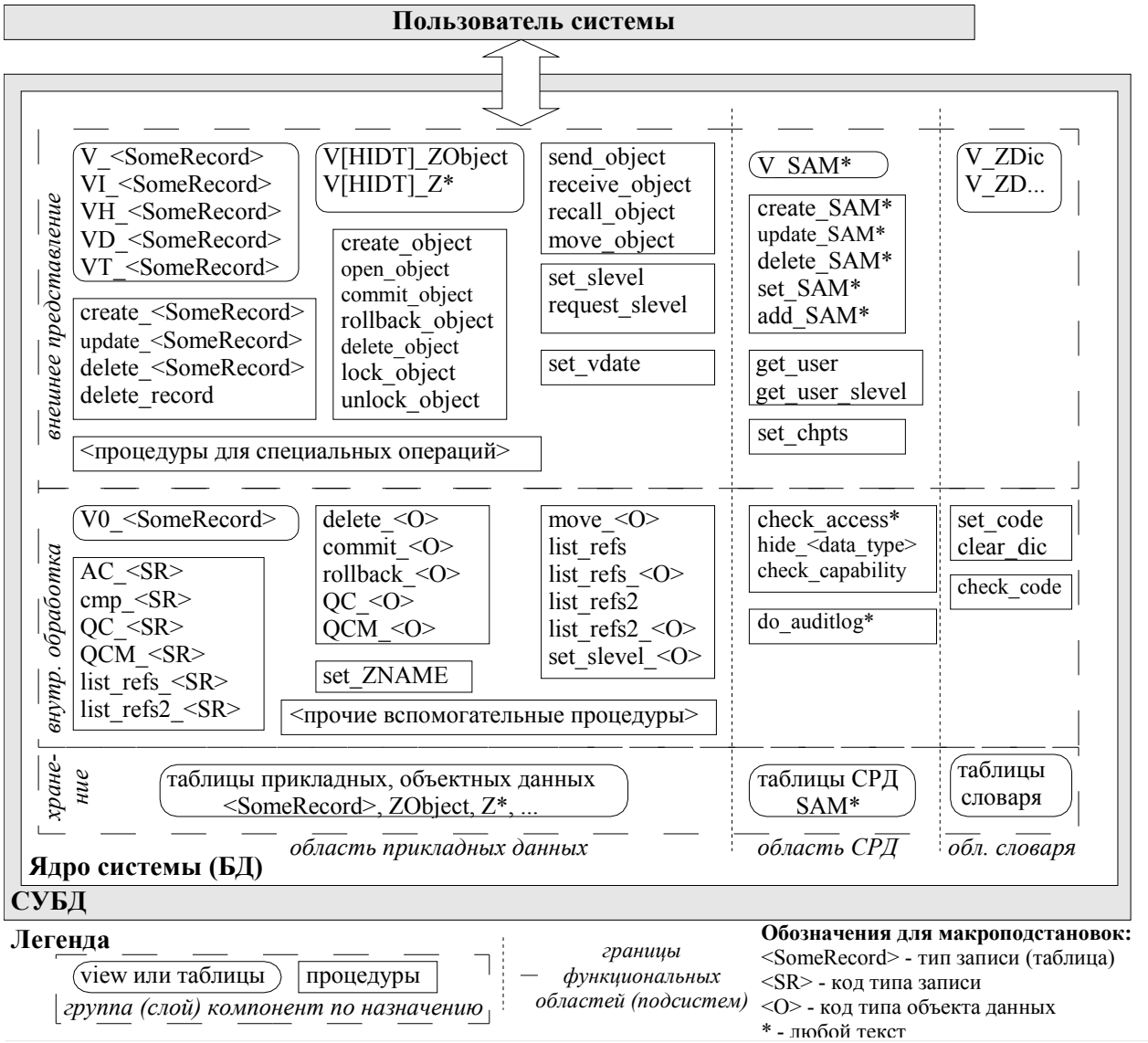


Рис. 7 Схема ядра системы

В двух предыдущих главах этой работы сформулированы базовые догмы, понятия, и описаны принципы, на которых предлагается строить целевые ИС. Эта глава посвящен следующему шагу от абстракции к конкретной реализации. При этом следует понимать, что конкретизация любой абстракции – лишает ее некоторой части потенциальных свойств, и одновременно наделяет новыми, затеняющими простоту исходной конструкции. С другой стороны, абсолютно абстрактное описание – абсолютно непонятно и бесполезно для абсолютного большинства возможных читателей. Помимо понятности, есть серьезная проблема жизнеспособности различных реализаций, созданных на основе одной и той же исходной модели. По этой причине, даже для основных понятий, описанных ранее, была предпринята попытка найти компромисс между абстракцией и реальностью, для которой эта абстрактная модель формулировалась.

Искомая «золотая середина» этого раздела – в минимальном объеме описать внутреннее устройство БД, с перечислением объектов, ее составляющих, а также предложить систему их именования (см Рис. 7). Этих данных еще недостаточно для немедленного начала реализации реальной системы на реальной СУБД, но они хорошо показывают путь, по которому можно пройти, и объем работы, который надо будет на нем проделать, для написания работоспособного *ядра системы*.

На схеме ядра системы (Рис. 7) четко обозначены три группы (слоя) объектов БД:

- *внешнее представление* – хранимые процедуры и view, доступные пользователю БД. Все эти объекты БД должны иметь встроенные механизмы идентификации действующего пользователя и разграничения доступа, в соответствии с правилами TIS/SQL подхода. При этом view позволяют только читать информацию, а ее изменение осуществляется только хранимыми процедурами.
- *внутренняя обработка* – процедуры и view, недоступные обычному пользователю, но предназначенные для использования другими процедурами. Для них допустимо передавать характеристики пользователя в качестве параметров процедуры, и не производить контроль прав доступа.
- *хранение* – к этому слою относятся только реляционные таблицы, в которых находятся реальные данные. Пользователю они недоступны, все манипулирование данными осуществляется только через объекты *внешнего представления*.

Подразумевается, что каждая хранимая процедура внешнего представления, если ее исполнение подразумевает изменение данных, должны исполняться в пределах отдельной *физической транзакции*. На практике это не всегда можно гарантировать средствами конкретной СУБД, и в этом случае обеспечение этого принципа возлагается на прикладное ПО. Проблемы, к которым приводит невозможность обеспечить соблюдение этого требования уже рассматривались ранее (напр. п. 9 в Таб. 1).

> **NB** Расположение слоев на Рис. 7 наводит на мысль, что объекты внешнего представления должны взаимодействовать с объектами слоя хранения только косвенным образом, через слой «внутренняя обработка». В действительности это не так, строгого ограничения на непосредственный доступ любых объектов к любым другим объектам БД нет, но следует жестко придерживаться принципа: «если для совершения некоторого действия имеется процедура внутренней обработки, то ее неиспользование какой-либо другой процедурой – должно быть обоснованным и хорошо документированным».

Все процедуры внешнего представления, в вопросах идентификации пользователя и контроля прав доступа, полагаются на соответствующие процедуры подсистемы СРД (`get_user`, `get_user_slevel`, `check_access*`, `check_cpability`). View внешнего представления обычно полагаются на эти процедуры только в вопросах идентификации пользователя (`get_user`, `get_user_slevel`) и в вопросах сокрытия содержимого защищаемых полей (`hide_<datatype>`); в остальном они используют содержимое таблиц SAM непосредственно, что обусловленно вопросами производительности.

Процедуры *внешнего представления*, обслуживающие *область прикладных данных*, можно разделить на три категории: действия над объектами данных, действия над записями определенных типов и специальные операции.

- Первая категория – это процедуры с фиксированными именами (`commit_object`, `delete_object` и т.п.), обычно являющиеся своеобразной оболочкой к переключателю вызовов на процедуры внутренней обработки, выполняющие реальную обработку для конкретных типов ОД (`commit_<O>`, `delete_<O>`). Часть процедур первой категории (`create_object`, `open_object`, `lock_object` и т.п) не требует никаких знаний о реальной структуре ОД, и не нуждается в объекто-специфичных процедурах внутренней обработки. Все процедуры этой категории пишутся вручную.
- Вторая категория – это шаблонные процедуры, в именах которых четко сказано, для какого типа записей они предназначены (`(create|update|delete)_<SomeRecord>`). Эти процедуры сами выполняют всю необходимую работу, обращаясь к процедурам внутренней обработки только за вспомогательными операциями. Код этих процедур должен порождаться кодогенератором, на основе описаний таблиц (подобных Таб. 18).
- Третья категория – специфические операции, не укладывающиеся в стандартную схему обращения с объектами данных. Примером процедур этой категории будут операции построения отчетов. Естественно, все эти процедуры требуют ручного кодирования.

> **NB** Процедура `delete_record` – является переключателем к записе-специфичным процедурам и относится к первой категории. На ее примере удобно обратить внимание на то, что минимизация количества хранимых процедур внешнего представления, необходимых для пользовательского приложения, позволяет, в

некоторых условиях, существенно сэкономить ресурсы сервера СУБД. Дело в том, что для вызова хранимой процедуры приложение должно создать т.н. «Prepared Statement» - объект, обеспечивающий взаимодействие с сервером СУБД для передачи параметров и получения результатов вызова. Использование заранее подготовленного Preapared Statement для всех обращений к этой процедуре, в пределах данной сессии с БД, позволяет существенно поднять производительность (в предельных случаях в 10 или даже 100 раз), поэтому целесообразно задуматься о кэшировании этих объектов, вместо разрушения после каждого вызова. Однако здесь есть скрытая проблема – во многих СУБД, Prepared Statement, имеет свое отражение на стороне сервера БД, потребляющее некоторые ограниченные ресурсы. Использование одной процедуры на все типы записей – очень существенная экономия этих ресурсов. Еще одним способом экономии может стать использование переключателя `edit_<SomeRecord>` к процедурам `create_<SomeRecord>` и `update_<SomeRecord>`.

Процедуры внутренней обработки можно разделить на те же три категории, но со следующими поправками:

- Большая часть процедур обработки ОД уже не имеет фиксированных имен, но содержит в своем имени указание на тип ОД, для работы с которым предназначена. Эти процедуры более шаблонны, и в большом проекте их построение можно возложить на кодогенератор.
- Для процедур работы с записями остались только вспомогательные задачи. Код некоторых из них не может быть выполнен кодогенератором. Прежде всего – это процедуры «ручного» *контроля качества* QCM_*. (Впрочем – без таких «рукописных» процедур можно обойтись, по крайней мере на стадии первой условно-работающей внутренней версии).
- прочие вспомогательные процедуры – это уже не только и не столько комплексные, проблемо-ориентированные операции, но разные вспомогательные мелочи вроде проверок значений полей на соответствие некоторым критериям или функций различных преобразований.

Индивидуальные особенности, а также назначение всех процедур, обозначенных на схеме, будет дано далее, но прежде необходимо сказать пару слов о языках программирования, на которых они должны быть написаны. За незначительными исключениями все СУБД, пригодные для TIS/SQL подхода, обладают своим собственным вариантом процедурного языка SQL, предназначенного для написания хранимых процедур, функций и триггеров (TransactSQL у MS-SQLServer и Sybase SQLServer; PL/SQL – Oracle; SQL/PL – DB/2; PL/PgSQL – PostgreSQL; PSQL – FireBird & InterBase). Общая идея всех этих языков – дополнить SQL необходимым для превращения его из языка запросов в язык программирования, но реализация этого – различна, и между собой они несовместимы. Злые языки упрекают их в ограниченности и в неисчислимых недостатках из нее вытекающих, но для решаемой задачи эта ограниченность является скорее благом – код процедур получается простым, коротким и линейным, что облегчает его понимание, верификацию и сертификацию. Ограниченность возможностей языка минимально-достаточными для решаемой задачи – предохраняет программиста от совершения ошибок, вызванных незнанием или непониманием всех возможностей языка и их побочных проявлений. Для такой критичной части системы, как ядро, отвечающее за сохранность и достоверность информации – лучше уложить всю необходимую функциональность в минимуме кода (по крайней мере на стадии прототипа).

Все, перечисленные ранее, СУБД имею возможность для подключения хранимых процедур и функций написанных на внешних языках. Их разумное использование способно решать многие возникающие проблемы, но здесь их применение не рассматривается – все необходимые для TIS/SQL процедуры могут быть написаны на процедурном SQL. При возникновении потребности переписать большую часть процедур на внешнем языке, следует рассмотреть переход на схему, когда ядро системы находится вне СУБД, и управляет ею, а не наоборот. Такая модель позволяет строить более сложные, функциональные и масштабируемые системы, но и затраты на их разработку будут на порядки выше. TIS/SQL подход инвариантен к взаимоотношениям подчиненности *ядра системы* и СУБД.

3.1. Служебные данные – SAM, Dictionary. Шаблоны процедур ведения данных.

Процедуры и, в меньшей степени, view, работающие со служебными данными, достаточно иррегулярны, и поэтому всегда пишутся вручную. Разработка подсистемы словаря (в минимальном объеме, обозначенном на Рис. 7) выполняется прежде всего остального. Следом за ней должна быть выполнена подсистема SAM, разработка которой позволяет отработать необходимые технические приемы программирования на данном диалекте, которые далее будут использоваться для всех остальных процедур. Ориентировочный объем работы по кодированию процедур СРД – 3-4 тысячи. строк – хорошо подходит для учебной или квалификационно-оценочной задачи. Впрочем, при разработке подсистемы SAM, можно начать с минимального набора заглушек, который позволит сразу начать работу над прикладной частью, не дожидаясь готовности всей СРД.

Подсистема словаря состоит всего из одного view внешнего представления и трех процедур внутренней обработки. View `V_ZDic` отображает таблицу `ZCode` без каких либо модификаций. Хранимая процедура `clear_dic` – удаляет все содержимое словаря (или некоторую его часть), а процедура `set_code(dic,code,value,descr)` – создает или модифицирует код 'code' в словаре 'dic' (т.е. фактически – одну строку в таблице `ZCode`, см Таб. 17). Обе эти процедуры доступны только для администратора БД и используются только в скриптах загрузки и обновления словаря. Хранимая процедура `check_code(dic,code)` проверяет наличие кода code в словаре dic, и используется всеми остальными процедурами, для проверки допустимости значений словарных полей.

Подсистема SAM состоит из 9-ти view внешнего представления, по одному на каждую таблицу (см Рис. 4); некоторого количества процедур ведения данных в этих таблицах (по 2 или 3 процедуры на таблицу); а также процедур, по своим функциям не связанных с ведением данных, или не вписывающихся в стандартную модель по-записного редактирования (создать/изменить/удалить). Последние можно разделить на следующие группы:

- *идентификация* действующего пользователя – функция get_user() возвращает идентификатор действующего пользователя (`SAMUser.id`), а `get_user_slevel()` - его уровень доступа. Все остальные процедуры и view полагаются на эти две функции, составляющие необходимый минимум для начала работы над процедурами прикладной области.
- манипулирование состоянием областей данных – на схеме обозначена всего одна функция из этой группы, `set_chpts`, манипулирующая значением поля `SAMScopePolicy.chpts`. Ее минимальная функциональность – установить значение этого поля в текущие дату/время, что гарантирует неизменность этого среза истории локальной *области данных*.
- проверка прав доступа действующего пользователя – группа функций `check_access*` (check_access_write, check_access_read, check_access_history т.д.). Используется всеми остальными процедурами для проверки прав действующего (или – указанного) пользователя на совершение запрошенных операций над объектами доступа (ОД, записью, полем). В качестве характеристик объекта доступа, им передаются: идентификатор области (ZSID), идентификатор ОД (ZOID) и *код типа* (ОД, записи или поля). Группа функций `hide_<data_type>` - предназначена для использования во view, с целью сокрытия значений полей, неразрешенных к чтению действующему пользователю. Идея такова – функция получает значение и характеристики объекта доступа, и возвращает – это же значение, если пользователь имеет право на чтение указанного поля, или NULL – если не имеет. Функция `check_capability(ZSID, cap_code)` – проверяет наличие у пользователя привилегии 'cap_code' на область данных ZSID и возвращает TRUE или FALSE.
- ведение *журнала аудита* – процедуры `do_auditlog*` осуществляют добавление записей в журнал аудита. Различия между процедурами этой группы – косметические, для начала можно обойтись одной единственной процедурой do_auditlog(весь набор полей таблицы `SAMAuditEvent`).

Для подсистемы SAM, отнесение процедур к внешнему представлению или внутренней обработке весьма условны, практически все перечисленное можно, при необходимости, сделать доступным пользователю. Исключение составляют только процедуры do_auditlog, где потребуется написать отдельную версию для внешнего представления, чтобы пользователь не мог писать в *журнал аудита* что попало.

Процедуры ведения данных могут быть написаны по нескольким шаблонам, в зависимости от семантики обслуживаемого понятия. Если пренебречь вариациями, то основных шаблонов три – *create/update/delete*, *add/remove* и *set/delete*.

- Базовый шаблон – *create/update/delete* – подходит для объекто-подобных сущностей, имеющих уникальные числовые идентификаторы, порождаемые при их создании. Такими сущностями являются SAMUser, SAMGroup и SAMScope. В этом случае, процедуры create и update имеют в качестве параметров все поля записи, за исключением параметра id, который передается только в процедуру update, где однозначно указывает на запись, которую нужно обновить. В процедуре create осуществляется порождение нового уникального id для создаваемой записи, который возвращается пользователю, после успешного выполнения. В процедуру delete передается только один параметр – уникальный идентификатор id. Разновидностью этого шаблона будет использование вместо двух процедур create и update, процедуры-переключателя edit.
- Шаблон *add/remove* хорошо подходит для сущностей, у которых все поля входят в первичный ключ (см SAMUserGroup, SAMUserCapability). В этом случае параметры у обоих процедур – одинаковые, но одна из них запись создает, а другая – удаляет. Разновидностью этого шаблона является добавление записей, не предполагающих их изменение или удаление (как, например, `SAMAuditEvent`).
- Последний из шаблонов, *set/delete*, ориентирован на сущности с составным первичным ключом и группой описательных полей (см SAMScopePolicy, SAMACL*). Процедура set похожа на процедуру update, за тем исключением, что не найдя подходящей записи – она ее создаст. Процедура delete получает в качестве параметров первичный ключ записи, при этом можно реализовать ее с учетом передачи неполного ключа, для удаление группы записей.

Для всех процедур ведения данных целесообразно использовать один и тот же набор возвращаемых параметров, включающий идентификатор созданной записи, код ошибки и сообщение об ошибке. Даже если какие-то из этих параметров не будут использоваться конкретной процедурой, унификация структур и алгоритмов обработки результатов выполнения окупит некоторую избыточность. В листинге, приведенном в приложении В, для возврата результатов выполнения функции используется следующий тип данных, которой можно взять за основу:

```
CREATE TYPE retstatus AS (id int, errcode int, errdesc varchar(255));
```

3.2. Структура таблиц, составляющих ОД. Таблицы ZObject, ZName.

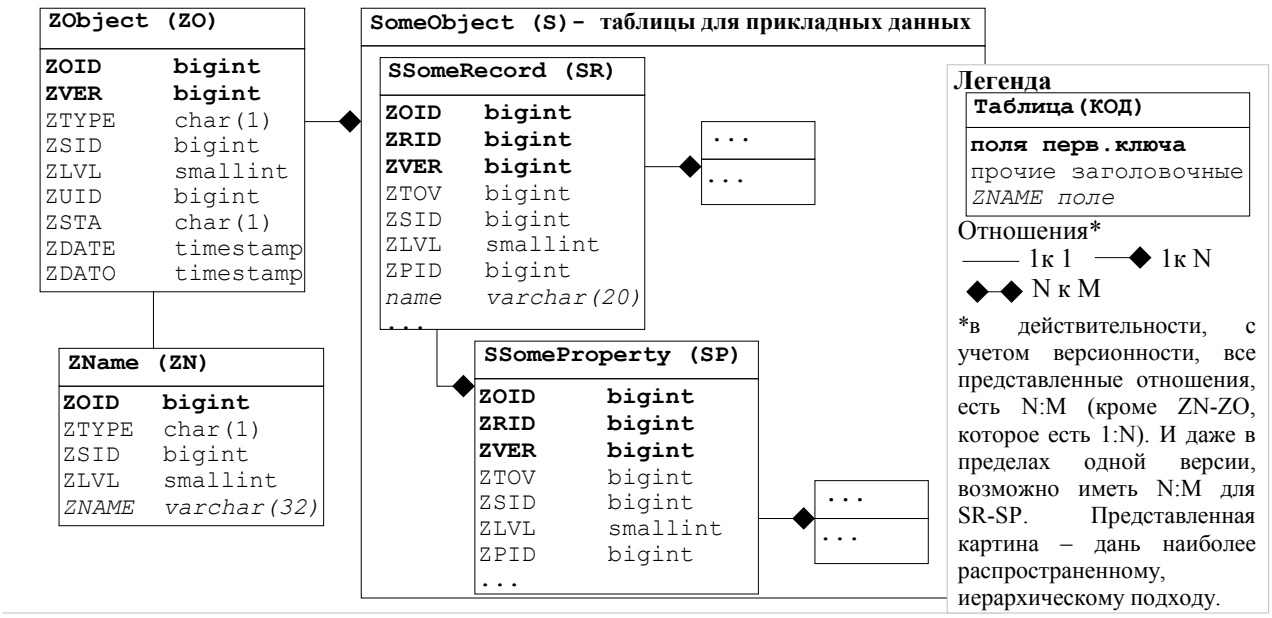


Рис. 8 Структура таблиц объекта данных

Таблицы, используемые для хранения объектов данных, можно сразу разделить на две группы:

- системные таблицы, чьи имена начинаются с префикса «Z», и которые используются для всех типов ОД.

- прикладные таблицы, чьи имена начинаются с кода типа ОД, и которые используются только для объектов этого типа.

Большая часть структур, использованных на Рис. 8, подробно описаны ранее по отдельности: главная служебная таблица `ZObject` – см. Таб. 2 (стр. 18); стандартный заголовок, используемый во всех прикладных таблицах – см. Таб. 4 (стр. 19). Здесь изображена та действительная структура ОД, которая подразумевается при составлении квази-ER диаграмм, но не отображается на ней (см Рис. 5, стр. 40). Новой является только индексная таблица `ZName`, уже упоминавшаяся, не еще не описанная подробно.

Проблема, приведшая к возникновению таблицы ZName такова: большинство типов ОД имеют заголовочную запись, содержащую одно или несколько полей, образующих уникальный идентификатор документа – имя или номер, под которым он известен пользователям, и который используется в делопроизводстве и во внешних системах. Естественным образом возникает потребность быстро находить объекты по этому идентификатору, т.е. по этим полям необходимо построить индекс, а в процедурах контроля качества реализовать контроль уникальности этого «имени». Здесь может вскрыться следующая проблема:

1. Пользователи желают, чтобы в БД «имена» хранились с сохранением регистра: «MyName» «yourname», «hIsName», «hErnAme».
2. Поиск и контроль уникальности необходимо осуществлять независимо от регистра.
3. СУБД этого обеспечить не может.

Проблема регистра взята как наиболее распространенная и понятная, в действительности она лишь частное проявление проблемы функциональных индексов по необратимым функциям от значений полей. Решением ее является введение специальной *индексной таблицы*, хранящей значение этой функции, по которому можно построить индекс. (Вместо таблицы можно ввести специальное индексное поле в той же таблице, но это не всегда удобно и целесообразно.)

Таким образом, таблица ZName является общим, для всех типов ОД, хранилищем уникальных «имен» документов, тем самым добавляя некоторое новое свойство к понятию *объект данных*. Все заголовочные поля дублируются из таблицы ZObject, а значение поля ZNAME – есть функция (по умолчанию – приведение к верхнему регистру) от некоторого поля «имя» в заголовочной записи ОД. Запись в этой таблице, для конкретного ОД, присутствует только если наличие такого «имени» предусмотрено его структурой данных, и если это «имя» не NULL. Поскольку данные в индексных таблицах вторичны, то хранение истории их изменений не предусмотрено (впрочем – это не догма).

Пространство, на котором обеспечивается уникальность ZNAME, зависит от конкретных потребностей. Наиболее очевидна уникальность в пределах всех ОД этого типа, или в пределах всех ОД этого типа в конкретной *области данных*.

> **NB** В распределенных и реплицируемых системных возможно обеспечить уникальность только в пределах локальных *областей данных* конкретной инсталляции.

Не исключено, что в процессе эволюции поле ZNAME войдет в структуру таблицы ZObject. Возможно, что от него вообще можно будет отказаться, но для данной работы это не целесообразно – наличие таблицы ZName, демонстрирует способ (идею) решения целого класса задач.

Еще одна проблема, для решения которой можно использовать индексные таблицы, связана с самой сложной и самой запутанной частью любой реляционной СУБД – оптимизатором запросов. В некоторых случаях, большое количество индексов на таблицу мешает ему работать, а перенос некоторого их количества в отдельную таблицу – способно сгладить эту проблему.

> **NB** Если права доступа на заголовочную запись ОД отличаются от прав доступа на сам ОД, то рекомендуется разграничивать доступа к данным ZObject и ZName одинаковым образом, на основе прав доступа к ОД.

Еще одна, заслуживающая внимания, идея связана с некоторыми крайними способами нормализации структуры БД. Для описания этой идеи необходимо вернуться к квази-ER диаграмме (Рис. 5, стр. 40), и внимательно посмотреть на сущности `DRow` и `DProperty`, с учетом следующих пояснений:

1. Представим, что в таблице DRow необходимо иметь несколько десятков описательных полей, большая часть которых остается незаполненной.
2. Усилим проблему постоянно возникающими потребностями в изменении состава этих полей.

3. В этом случае, целесообразно произвести нормализацию, выделив все эти поля в отдельную таблицу (см. `DProperty`), где каждая строка будет эквивалентом одного такого поля.
4. Поле `type`, в такой таблице, – словарное, и является заменой имени поля, а поля `nvalue` и `tvalue` – позволяют хранить числовое или символьное значение (или некоторое комплексное, состоящее из числа и текста), в зависимости от `DProperty.type`.
5. Для хранения описаний типов полей целесообразно завести специальную таблицу словаря, в которой по каждому полю будут указаны – код типа, название, типа данных, длина и т.п. Зачастую, одной таблицей словаря ограничиться не удается.

У рассмотренного приема есть достоинства и недостатки, большая их часть либо лежит на поверхности, либо хорошо описана в литературе по реляционным СУБД, но TIS/SQL подход добавляет к ним ряд своих, специфических, связанных с разграничением доступа. До тех пор, пока политика доступа к этим *полям-свойствам* одинакова – проблем не возникает, но если потребуется иметь разные права доступа, в зависимости от типа «свойства», то описанные ранее способы разграничения доступа и структура ACL (см раздел 2.3.9.3, на стр. 31) потребуют определенной доработки.

Прежде всего следует напомнить, что основным способом разграничения доступа, в TIS подходе, является назначение прав на *тип объекта данных*, *тип записи* или *тип поля*, в *области данных*. До тех пор, пока поля были полями в таблице, они попадали под понятие «*тип поля*», и для каждого из них можно было ввести отдельную политику разграничения доступа. После того, как все поля стали записями одного и того же типа – эта возможность утрачена. Чтобы ее восстановить, надо разработать механизм назначения прав доступа, различающий, в пределах одного ОД, индивидуальные записи одного типа, в зависимости от значения некоторого поля. В случае таблицы `DProperty`, это поле `type`.

Однако привязка разграничения доступа к прикладным данным – противоречит одному из догматов, положенных в основу TIS подхода. Решением этой коллизии будет перенос поля `type` в стандартный заголовок `STD_HEADER` (Таб. 4), под именем `ZFLD`, с присвоением новой структуре заголовка названия `STD_HEAFLD`.

После этого, необходимо либо привести систему кодирования ZFLD к общему стандарту кодирования *типов объектов*, *типов записей* и *типов полей*, либо – разработать для этого случая отдельную структуру ACL.

3.3. Ссылки между объектами

Вопросы установления ссылок между *объектами данных* уже рассмотрены ранее, в разделе 2.3.4 (стр. 20). Часть проблем, связанных с использованием ссылок, становиться понятной после прочтения раздела 2.3.8 (стр. 24), посвященному понятию «*транзит*». Если попытаться резюмировать в двух предложениях, все что сказано о ссылках ранее, то получиться так:

- Ссылки устанавливаются с применением уникальных идентификаторов ОД ZOID таким образом, что ссылающийся ОД, в своих прикладных данных, имеет поля, используемые для хранения идентификаторов тех ОД, на которые он ссылается.
- При некоторых операциях над ОД (например, при перемещении) необходимо получить идентификаторы всех, на кого он ссылается, или кто ссылается на него.

Для получения таких списков ОД предназначены процедуры `list_refs*`. Процедура, содержащая в своем имени *код типа ОД*, выполняет реальную работу по получение списка связанных объектов, а процедуры с фиксированными именами – принимают код типа ОД в качестве параметра, и выполняют роль «переключателя». Процедуры, содержащие в своем имени *код типа записи*, выполняют получения списка *объектов данных*, на основании ссылок, содержащихся в записях соответствующего типа.

Процедуры `list_refs(ZTYPE, ZOID)` и `list_refs_*(ZOID)` - готовят список ОД на которые ссылается указанный *объект данных*, а процедуры `list_refs2(ZTYPE, ZOID)` и `list_refs2_*(ZOID)` – список тех, которые сами ссылаются на него. Каждый возвращаемый ZOID снабжается кодом, показывающим «качество» ссылки (например: C – контейнер, E – вложение, R – простая ссылка и т.п.).

Если система предполагает интенсивное движение ОД между *областями данных* или иную активность, требующую постоянного контроля или анализа ссылок, то целесообразно реализовать

индексную таблицу `ZRef`, в которой будут содержаться все действующие ссылки между *объектами данных*. В Таб. 19 приведен минимальный вариант такой таблицы.

Таб. 19 ZRef (ZR) – Список ссылок ОД (индексная таблица)

№	Поле	Тип	Описание
01	**ZOID**	bigint	уникальный идентификатор ОД
02	**ZVER**	bigint	уникальный номер версии ОД
03	ZTYPE	char(1)	код типа ОД
04	ZSID	bigint	идентификатор *области данных*, которой принадлежит ОД
05	ZLVL	smallint	уровень секретности ОД (используется мандатной СРД)
06	ZOID2	bigint	ID *объекта данных*, на который имеет ссылку ОД ZOID
07	ZREF	char(1)	тип ссылки (напр. E – вложение, R – простая ссылка и т.п.)

3.4. Таблицы данных и система view (V_*, V{HID0T}_*)

Исходная парадигма для реализации TIS в структуре реляционной БД предполагает, что каждой табличной сущности, изображенной на квази-ER диаграмме, соответствует одна одноименная таблица и набор view, чьи имена образуются приписыванием к имени сущности некоторого префикса. Таблица хранит все версии всех записей (прошлое, настоящее и возможное будущее), а view отображают разные срезы этого массива данных, накладывая на отображаемую информацию фильтры, учитывающие права доступа конкретного пользователя. Назначение каждого view кодируется в его префиксе, точнее говоря в части префикса, заключенной между обязательным начальным символом «V» и обязательных конечным символом «_». Вот их список:

- V_ - все записи (версии записей), действительные в данный момент времени (ZTOV=0). Это основное view для интерактивной работы с системой. Дискреционной СРД права доступа контролируются по значению поля `SAMACL*.reada` (см Таб. 11 на стр. 31).
- VH_ - все версии всех записей, кроме относящихся к незавершенным логическим транзакциям. Это view предназначено для анализа истории изменений. Права доступа контролируются по `SAMACL*.historya`. Для ZObject возможно приравнять VH_ к V_, чтобы информация о существовавших версиях ОД была составной частью его актуального состояния.
- VI_ - записи из незавершенных *логических транзакций*, осуществляемых действующим пользователем. Позволяет пользователю восстановить контроль над *логической транзакцией*, без потерь уже введенных данных, например после аварийного завершения клиентского приложения.
- VD_ - версии всех записей, которые были действительны в указанный пользователем момент времени . Срез задается вызовом процедуры `set_vdate(slice)`. VD_ должно отображать то же самое состояние таблицы, что отображало view «V_» в заданный момент времени. Права доступа контролируются также как для VH_.
- V0_ - все записи, действительные в данный момент времени (ZTOV=0), без фильтрации СРД. Предназначено для использования хранимыми процедурами. С применением этих view можно оценить потери производительности на фильтрацию СРД. В дополнение к «V0_», можно реализовать view «VD0_», которое будет отображать заданный исторический срез, без фильтрации СРД. С применением системы view «VD0_», возможно реализовать хранимые процедуры получение отчетных документов, независящие от прав доступа действующего пользователя. Для разграничения доступа к этим таким процедурам системы, можно использовать механизм привилегий (см раздел 2.3.9.4, стр 33)
- VT_ - все записи, принадлежащие *объектам данных*, находящимся в транзите, по отношению к которым пользователь имеет возможность осуществить операцию «принять». Права доступа контролируются по `SAMACLScope.reada`, но не для той области, в которой еще находится ОД, а для той, в которую он отправлен.

При написании SQL запроса, на основе которого создается view, необходимо учитывать способность оптимизатора запросов конкретной СУБД справляться с конкретными SQL конструкциями. Вот пример SQL оператора создания view для таблицы SSomeRecord (Рис. 8), от которого можно отталкиваться:

```
CREATE VIEW V_SSomeRecord AS SELECT * FROM SomeRecord WHERE ZTOV=0 AND (
```

```
ZSID IN (SELECT sid from SAMACLScope WHERE obj_type = 'S' reada = 'Y' and
 gid in ( select id from SAMUserGroup where uid = get_user() )
OR
ZOID IN (SELECT ZOID from SAMACLObject WHERE obj_type = 'S' reada = 'Y' and
 gid in ( select id from SAMUserGroup where uid = get_user() )
)
```

Иногда, может возникнуть потребность разделить единую таблицу, содержащую все версии записей, на две или три, чтобы устаревшие данные не мешали действующим, или работа в пределах логической транзакции не зависела от основного массива данных. Для именования таких таблиц можно разделить префиксы между view и таблицами следующим образом: «V» - показатель view; «H», «I» - показатели таблиц; «_» - разделитель между именем табличной сущности и префиксами. Тогда: SSomeRecord - таблица с действующими данными; H_SSomeRecord – с устаревшими; I_SSomeRecord – с данными незавершенных транзакций.

3.5. Хранимые процедуры манипулирования ОД

Принципы манипулирования *объектами данных* унаследованы от аналогичных принципов обращения с файлами в современных операционных системах. Любое изменение ОД начинается с его «открытия» (перевода из «свободного» состояния, в состояние «открыт для редактирования»). В результате «открытия» ОД, субъект получает некоторый параметр, идентифицирующий это состояние и предъявляемый далее при вызовах всех процедур, в рамках этой *логической транзакции*. Завершает логическую транзакцию операция «закрытие», осуществляющая актуализацию всех внесенных изменений или их отмену.

Параметром, идентифицирующим «открытое» состояние ОД, является совокупность значений {ZOID,ZVER}. Операция «открыть» сопровождается порождением новой версии ОД, номер которой возвращается субъекту, а родственная ей операция «создать» - порождает первую версию нового ОД и возвращает его идентификатор ZOID. Все процедуры, помимо специфичных для них результатов, всегда возвращают код завершения (0 – в случае успеха, иначе – код ошибки) и сообщение об ошибке.

Для дальнейшего рассмотрения, операции над ОД удобно разбить на четыре группы: *редактирование*, *блокировка*, *перемещение*, *изменение уровня секретности*. Процедуры, обслуживающие *редактирование,* таковы:

- `create_object(SZID, ZTYPE, ZLVL)` – создает новый ОД типа `ZTYPE`, в области данных `ZSID`, с уровнем секретности `ZLVL` (если ZLVL – NULL, то с минимально допустимым в данный момент). Возвращает ZOID нового ОД и присвоенный ему ZLVL. Созданный ОД находится в открытом состоянии, его версия: ZVER=1.
- `open_object(ZOID,ZVER)` – открывает ОД ZOID. Если ZVER не NULL, то операция будет выполнена только если последняя версия ОД соответствует указанной. Использование параметра ZVER позволяет гарантировать, что документ не изменялся с тех пор, как мы начали с ним работать, и является альтернативой парадигме явных блокировок, которые столь же явно должны сниматься прикладным ПО, что не всегда возможно обеспечить. Процедура возвращает новый ZVER.
- `rollback_object(ZOID,ZVER,force_user)` – отменяет все внесенные изменения и «закрывает» ОД. Параметр `force_user='Y'` задействует привилегию FORCE_ROLLBACK (Таб. 13), что позволяет завершать *логические транзакции*, начатые другими пользователями. Реальную работу выполняет процедура внутренней обработки `rollback_<O>`, соответствующая типу ОД.
- `commit_object(ZOID, ZVER)` – выполняет процедуры *контроля качества* состояния ОД, с учетом внесенных изменений; в случае успеха – «закрывает» ОД и делает актуальной его новую версию. Для работы с конкретным типом ОД используются процедуры внутренней обработки `commit_<O>`, `set_ZNAME`, `QC_<O>` и `QCM_<O>` («QC» – Quality Control, «M» – Manual). Разница между процедурами контроля качества QC* в том, что `QC_<O>` может порождаться кодогенератором, а `QCM_<O>` - всегда пишется «вручную».
- `delete_object(ZOID, ZVER)` – удаляет ОД. Операция осуществляется только над «закрытым» и «незаблокированным» ОД. Параметр ZVER выполняет ту же функцию, что и для `open_object()`. Действительная работа выполняется соответствующей процедурой delete_<O>.

Процедуры блокировки и разблокировки ОД (`lock_object` и `unlock_object` соответственно) являются упрощенными вариантами `open_object` и `rollback_object`, за тем исключением, что в заблокированный ОД нельзя вносить изменений. Может оказаться полезным реализовать вариант процедуры `open_object`, позволяющий субъекту перейти от заблокированного состояния ОД к «открытому», минуя «разблокировку» и возможность перехвата инициативы другим субъектом.

Процедуры перемещения `send_object` и `move_object` требуют «свободного» состояния ОД, при этом move_object операция атомарная, а send_object – переводить ОД в состояние «заблокирован для перемещения» и требует последующего вызова одной из процедур завершающих перемещение (`receive_object` или `recall_object`). Параметр ZVER имеет то же значение что и для `open_object`. Вместе с указанным ОД перемещаются или блокируются все ОД, вложенные в него прямо или косвенно. Обработка конкретных типов ОД осуществляется процедурами `move_<O>` и `list_refs*`. Характеристики процедур перемещения таковы:

- `move_object(ZOID,ZVER,ZSID,with_history)` – переместить объект ZOID в область ZSID. Параметр with_history='Y' выполняет перемещение ОД вместе со всей историей изменений.
- `send_object(ZOID,ZVER, ZSID)` – помещает ОД в транзит, с указанием в качестве места назначения области данных ZSID.
- `receive_object(ZOID,ZVER, ZSID)` – завершает перемещение ОД ZOID, находящегося в транзите, в область ZSID.
- `recall_object(ZOID,ZVER, ZSID)` – отменяет незавершенное перемещение ОД ZOID, находящегося в транзите, в область ZSID.

Последняя группа операций над ОД – изменение уровня секретности – содержит две процедуры: `request_slevel` и `set_slevel`. Набор параметров у них идентичен: (ZOID, ZVER, ZLVL, with_history). Процедура `request_slevel` блокирует ОД и помещает его в транзит, с указанием запрошенного ZLVL. Процедура `set_slevel` производит изменение ZLVL, используясь для этого подходящую процедуру `set_slevel_<O>`. Назначение параметра ZVER, в основном, аналогично описанному для `open_object`, однако для выполнения `set_level` над ОД, находящимся в *транзите* вследствие `request_slevel`, ZVER должен строго соответствовать этой транзитной версии.

3.6. Хранимые процедуры редактирования и обработки записей ОД

Процедуры ведения записей ОД построены по шаблону *create/update/delete* (раздел 3.1, стр. 47). Все они доступны только в пределах *логической транзакции*, начатой процедурами `open_object` или `create_object`. В качестве обязательных параметров, идентифицирующие открытую версию ОД, передаются {ZOID,ZVER}. После обязательных параметров, в процедуры `create` и `update`, передаются значения для всех полей целевой таблицы. Помимо прочего, эти процедуры всегда возвращают код завершения и сообщение об ошибке.

- `create_<SomeRecord>(ZOID,ZVER,...)` – создает новую запись типа SomeRecord в ОД ZOID и возвращает ее ZRID.
- `update_<SomeRecord>(ZOID,ZVER,ZRID,...)` – обновляет запись ZRID в ОД ZOID.
- `delete_<SomeRecord>(ZOID,ZVER,ZRID)` – удаляет запись ZRID в ОД ZOID и все записи, прямо или косвенно подчиненные ей, в рамках жестких иерархических отношений ZRID-ZPID.

Процедуры create и update во многом похожи. В процессе кодирования, последняя получается из первой после доработки, увеличивающей ее объем и сложность в два раза (приблизительно). Обе они в вопросах контроля доступа и *контроля качества*, полагаются на один и тот же комплекс процедур внутренней обработки. Алгоритм обеих процедур линеен:

1. Проверить права доступа к запрошенной операции над записью.
2. Проверить права доступа к заполнению/изменению всех полей. Для этого используется процедура AC_<SR> («AC» – Access Control), возвращающая всем полям, к которым у пользователя нет доступа, их исходные значения (при создании – NULL, при изменении – из действующей версии).
3. Для операции update – сравнить новые значения полей со значениями в действующей записи ОД, и если все они идентичны – завершить работу, предприняв необходимые действия, предотвращающие появления версий, не имеющих значимых отличий. Для сравнения используется процедура cmp_<SR>.

4. Выполнить процедуры *контроля качества* QC_<SR> и QCM_<SR>; в случае неудовлетворительного результата – уведомит об этом пользователя и прекратить выполнение. Процедура QC_<SR> - порождается кодогенератором, а QCM_<SR> - пишется «вручную».
5. Создать новую запись или создать/обновить новую версию записи. Созданные или измененные записи остаются невидимыми для всех остальных пользователей до успешного выполнения процедуры `commit_object()`.

Еще две процедуры внутренней обработки необходимы для каждой прикладной таблицы (т.е. каждого *типа записи*) – `list_refs_<SR>` и `list_refs2_<SR>`. Эти процедуры предназначены для поиска и извлечения ссылок, содержащихся в таблице <SR>. Обе процедуры получают в качестве параметра ZOID объекта. Процедура `list_refs_<SR>` возвращает список ОД, на которые ссылаются записи, принадлежащие указанному ZOID. Процедура `list_refs2_<SR>` возвращает список ОД, которым принадлежат записи, ссылающиеся на указанный ZOID.

Программный код всех перечисленных процедур, за исключением `QCM_<SR>`, может порождаться кодогенератором, по формальным описаниям таблиц.

3.7. Необъектные пользовательские данные

Ранее (раздел 2.3.3 стр. 20) уже упоминалась возможность использования одно-записных сущностей в тех случаях, когда сложная структура *объектов данных* очевидно избыточна. Там же приводится вариант стандартного заголовка STD_HEANOR (Таб. 5). Теперь к этому необходимо добавить важное идеологическое замечание:

> **NB** При проектировании структуры БД, одно-записные сущности следует использовать только тогда, когда все ее данные «должны храниться в одной записи», а не просто «могут поместиться в одной записи».

Из этого следует, что необъектные сущности, по своей семантике, чем-то отличаются от документарных сущностей, описываемых с применением *объектов данных*, а значит и процедуры их редактирования могут требовать иного шаблон, чем *create/update/delete*. Некоторые идеи можно почерпнуть в разделе посвященном ведению служебных данных (3.1, стр. 47).

При использовании шаблона *create/update/delete*, процедуры ведения данных для каждой отдельной таблицы будут полностью аналогичны описанным в предыдущем разделе, за тем исключением, что они не потребуют параметров ZOID и ZVER. Впрочем, последний можно использовать в процедурах update и delete тем же образом, как его используют open_object и delete_object.

3.8. Таблица ZTransit

№	Поле	Тип	Описание
01	**ZOID**	bigint	идентификатор ОД в транзите
02	**ZVER**	bigint	номер версии ZOID (версия - блокирующая ОД для транзита)
03	ZSID2	bigint	идентификатор области данных, в котороую отправлен ОД
04	ZLVL2	smallint	запрошенный уровень секретности
05	ZCOID	bigint	ZOID контейнера самого верхнего уровня, для которого ZCOID=ZOID
06	ZCVER	bigint	номер версии ZCOID
07	ZSTA	char(1)	статус записи ∈ {T,A,O,W} (см Таб. 3)
08	ZDATE	timestamp	дата/время помещения в транзит
09	ZUID	bigint	идентификатор пользователя, поместившего ОД в транзит
10	ZDATO	timestamp	дата/время конца нахождения ОД в транзите
11	ZUIDO	bigint	пользователь, извлекший ОД из транзита

Таб. 20 ZTransit (ZT) – Объекты данных в транзите

1. В разделе 2.3.8 (стр. 24) введено понятие «*транзит*», и упомянута таблица `ZTransit`, его обслуживающая. В действительности, семантика этого понятия может существенно варьироваться в различных информационных системах, и может потребовать проектирования целого комплекса таблиц для его обслуживания. Здесь представлена простейшая структура одной таблицы (Таб. 20), которая позволит реализовать двух-шаговую операцию передачи одного ОД или группы ОД,

помещенных в один контейнер, из одной области данных в другую. Эта же структура может использоваться для двух-шаговой операции изменения уровня секретности. Приблизительный алгоритм отправки/приемки удобно описать на следующем примере:

1. Пускай в области данных A0 имеются ОД: X0{ZVER=2}; X1{ZVER=1};X2{ZVER=3}
2. Пускай X1 вложен в X0, а X2 в X1(т.е. оба они вложены в X0, но X1 – прямо, а X2 – косвенно)
3. Пользователь U1 выполняет send_object(X0,ZVER=2,ZSID=A1)
   3.1. X0 блокируется для перемещения, для чего в ZObject создается новая запись-версия:
   {ZOID=X0,ZVER=**3**,ZSTA='**T**',ZSID=A0,ZUID=U1}
   3.2. составляется список всех ОД, прямо или косвенно вложенных в X0:{X1,X2}; все они также блокируются для перемещения. Записи в ZObject:{
   {ZOID=X1,ZVER=**2**,ZSTA='**T**',ZSID=A0,ZUID=U1},
   {ZOID=X2,ZVER=**4**,ZSTA='**T**',ZSID=A0,ZUID=U1} }
   3.3. в таблице ZTransit создается три записи (по одной для каждого ОД): {
   {ZOID=X0,ZVER=3,ZSID=A1,ZCOID=**X0**,ZCVER=3,ZSTA='T',ZUID=U1,ZDATE=NOW}
   {ZOID=X1,ZVER=2,ZSID=A1,ZCOID=**X0**,ZCVER=3,ZSTA='T',ZUID=U1,ZDATE=NOW}
   {ZOID=X2,ZVER=4,ZSID=A1,ZCOID=**X0**,ZCVER=3,ZSTA='T',ZUID=U1,ZDATE=NOW} }
4. Теперь все три ОД находятся в транзите, при этом для X1 и X2 указано, что они попали в транзит не сами по себе, а по причине нахождения в объекте-контейнере X0 (ZCOID=X0)
5. Пользователь U2 выполняет receive_object(X0,ZVER=3,ZSID=A1)
   5.1. Для каждого из трех ОД, находящихся в транзите (X0, X1 и X2), выполняется соответствующая процедура внутренней обработки move_<O>, «перемещающая» записи в прикладных таблицах из области данных A0 в A1.
   5.2. записи в таблице ZObject, имеющие ZSTA='N', корректируются:
   {ZOID=X0,ZVER=2,ZSTA=**'M',**ZSID=A0,ZDATO=**NOW**}
   {ZOID=X1,ZVER=1,ZSTA=**'M',**ZSID=A0**,**ZDATO=**NOW**}
   {ZOID=X2,ZVER=3,ZSTA=**'M',**ZSID=A0**,**ZDATO=**NOW**}
   5.3. записи в таблице ZObject, имеющие ZSTA='T', корректируются:
   {ZOID=X0,ZVER=3,ZSTA=**'N',**ZSID=**A1**,ZUID=**U2**,ZDATE=**NOW**}
   {ZOID=X1,ZVER=2,ZSTA=**'N',**ZSID=**A1**,ZUID=**U2**,ZDATE=**NOW**}
   {ZOID=X2,ZVER=4,ZSTA=**'N',**ZSID=**A1**,ZUID=**U2**,ZDATE=**NOW**}
   5.4. записи в таблице ZTransit, корректируются:
   {ZOID=X0,ZVER=3,...,ZSTA='**A**',ZUIDO=**U2**,ZDATO=**NOW**}
   {ZOID=X1,ZVER=2,...,ZSTA='**A**',ZUIDO=**U2**,ZDATO=**NOW**}
   {ZOID=X2,ZVER=4,...,ZSTA='**A**',ZUIDO=**U2**,ZDATO=**NOW**}
6. Все три ОД переместились из области A0 в область A1.

Если вместо операции receive_object, пользователь выполнит recall_object, то завершение операции можно реализовать различными способами. Вот два варианта, для примера:

- Удалить все блокирующие записи в ZObject и все соответствующие им записи в ZTransit. При этом информация о том, что объекты данных находились в транзите, кто их туда поместил и кто отозвал – будет утрачена.
- Завершить операцию, как перемещение всех указанных ОД из области A0 в A0, с указанием в ZTransit: ZSTA='W' (отозвано).

Двух-шаговое изменение уровня секретности может выполняться аналогичным образом, с той поправкой, что такая операция не должна распространяться на вложенные ОД.

4. Архитектура прикладных приложений.

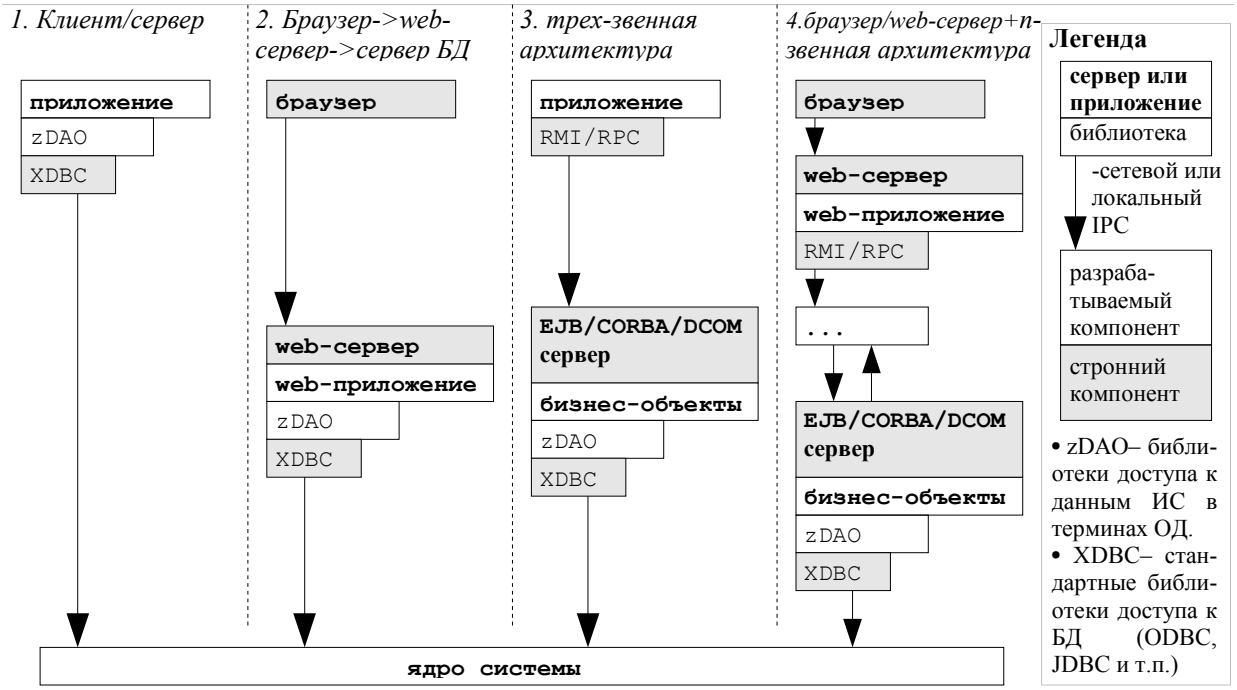


Рис. 9 Варианты архитектур ППО

Начиная эту главу, необходимо провести пограничную черту и подвести итог сказанному ранее. Фактически, изложение TIS/SQL подхода завершилось в предыдущей главе; все описанное ранее – есть фундамент (или каркас), на котором возводится ИС. Однако совершенно очевидно, что на этом строительство не заканчивается, а только начинается. Большая часть программного кода системы должна находиться вне ее ядра, взаимодействуя с ним через некоторый IPC механизм, предоставляемый СУБД, ОС или другим базовым ПО. Эта глава содержит рекомендации по построению прикладного ПО, обеспечивающего пользовательский интерфейс, а также по использованию промежуточных серверов приложений, позволяющих повысить защищенность и стабильность ИС, или интегрировать ее с другими системами (или в другие системы).

Рассматриваемые в этой главе идеи и конструкции, строго говоря, уже не являются эксклюзивной частью TIS/SQL подхода. Все они базируются на хорошо известных и достаточно распространенных сторонних технологиях, и являются своеобразной рекомендацией проектировщику и разработчикам, при построении целевой ИС, обратить внимание на них, и на предложенные идеи по использованию.

На Рис. 9 представлены четыре способа построения комплексов прикладного ПО, взаимодействующего с *ядром системы*. Эти варианты не взаимоисключающие, наоборот – все они могут сосуществовать параллельно, совместно используя не только ядро, но и значительную часть программного кода. На схеме они расположены в эволюционном порядке увеличения сложности:

1. *Клиент/сервер* – это простейший из используемых вариантов; именно в рамках этой идеологии сформировался TIS/SQL подход. В ней пользователь работает с системой через полноценное приложение, инсталлированное на его персональном компьютере и взаимодействующее с сервером БД напрямую. Каждый пользователь системы, также является и пользователем БД, что несет в себе определенный риск стабильной работе системы, обусловленный недостатками всех реальных СУБД (подробно см. далее).
2. *Браузер->web-сервер->сервер* БД – этот вариант архитектуры зародился и получил широкое распространение в Internet, как компромиссное решение, обусловленное спецификой этой среды. После существенной эволюции, развившей его достоинства и сгладившей недостатки, он распространился и на корпоративную среду, где стал серьезной альтернативой идеологии клиент/сервер. В этом варианте пользователь имеет на своем рабочем месте универсальную программу-браузер, посредством которой может подключаться и работать с любыми web-приложениями, имеющим пользовательский интерфейс из html/xml документов. Пользователь аутентифицируется web-

сервером, и прямого доступа к БД – не имеет. Его возможности по работе с системой ограничены функциями, заложенными в web-приложение, которое ему неподконтрольно, что существенно ограничивает возможность повлиять на работоспособность СУБД.

3. *трех-звенная архитектура* – представляет собой более изящный и идеологически проработанный, по сравнению с предыдущим, способ предоставить потребителю контролируемый доступ к предопределенной и ограниченной функциональности системы. В отличии от случая с web-приложением, бизнес-объекты под управлением сервера не предоставляют законченного пользовательского интерфейса, но лишь программный интерфейс, используемый другими приложениями (среди которых могут быть и web-приложения, и бизнес-объекты под управлением других серверов EJB, CORBA или DCOM). Принципиально, назначение промежуточного сервера аналогично назначению *ядра системы*, но при этом возможности языков программирования, на которых создаются бизнес-объекты, – на порядок превосходят любой процедурный диалект SQL. Таким образом, трех-звенная архитектура может стать альтернативой процедурному *ядру системы*, а TIS/SQL подход – просто TIS подходом, в котором та же система понятий и комплекс структур данных реализуются без интенсивного применения SQL (или вообще – без его использования).
4. *много-звенная архитектура* – является дальнейшим развитием трех-звенной, и возникает обычно в процессе интеграции нескольких ИС. В этом случает, вместо строгой иерархии потребитель/поставщик, возможно равноправное положение серверов приложений, когда каждый может обращаться к каждому.
5. Отдельным вариантом архитектуры, не обозначенным на Рис. 9, можно считать пакетные системы, построенные на принципах обработки очередей заданий (пакетов). TIS/SQL подход формировался в процессе работы над интерактивными системами, в которых все изменения поступают непосредственно в БД, но это не препятствует его использованию в качестве составной части, или даже в качестве основы пакетной системы. В современных условиях пакетные системы реализуются на тех же средствах, что и много-звенные, с активным использованием реляционных БД в качестве хранилища результатов обработки. Для хранения, передачи и архивирования пакетов, как и в предыдущем тысячелетии, применяются обычные файлы, поскольку ничего лучше и надежнее не придумано. Впрочем, семантика понятий «пакет» и «очередь», отличается от семантики понятий «файл» и «директория», обычно используемых для их реализации, и здесь может найти применение комплекс понятий, предлагаемый TIS подходом: *область данных*, *объект данных*, *ACL* и *привилегии*.

Как видно из Рис. 9, в основе всех вариантов лежит архитектура клиент/сервер. Однако, при использовании ее в «чистом» виде, каждый пользователь системы также является и пользователем БД, и, если тому нет специальных препятствий, может вместо штатного приложения воспользоваться любым другим, способным работать с СУБД (в т.ч. написанным самостоятельно). В этом кроется потенциальная угроза стабильной работе системы:

> **NB** Все реальные реляционные СУБД беззащитны перед атаками класса DoS (Deny of Service – «отказ в обслуживании») со стороны пользователя, допущенного к исполнению произвольных SQL запросов. Многие важные механизмы СУБД требуют определенных настроек среды, которые пользователь БД может нарушить, а администратор БД – упустить из виду. Помимо этого, все СУБД содержат известные и (пока) неизвестные ошибки, позволяющие захватить управление процессом сервера. Наиболее распространенный класс ошибок – примитивное «переполнение буфера» (buffer overflow) – настоящий бич всего современного ПО, написанного на языках C/C++. При этом, оперативная установка обновлений на СУБД – обычно затруднена, поскольку может нарушить устойчивую (устоявшуюся) работу системы.

Архитектура с сервером приложений – гораздо менее подвержена этой проблеме. Вариант с web-сервером позволяет существенно сэкономить на сопровождении ИС за счет централизованного расположения ПО и упрощения рабочих мест, а также на разработке пользовательского интерфейса – за счет использования возможностей, предоставляемых html/xml. Варианты с использованием EJB, CORBA или DCOM – более дорогостоящие в разработке – позволяют лучше структурировать ИС, что позволяет эффективнее разрабатывать и развивать сложные системы и/или эффективно задействовать большую команду разработчиков.

4.1. Библиотека zDAO

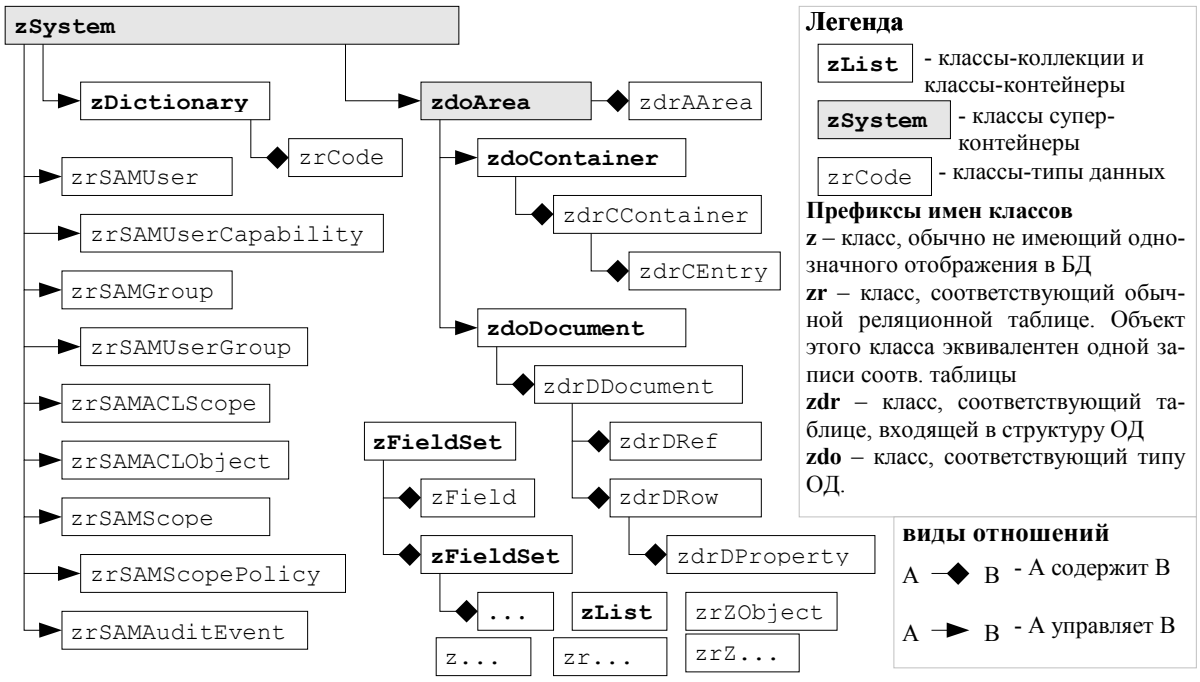


Рис. 10 Библиотека zDAO (с точки зрения управления)

Во всех вариантах архитектур на Рис. 9 присутствуют, как минимум, два общих компонента – библиотеки XDBC и zDAO. Первая из них — это условное обозначение любого механизма, выбранного для взаимодействия с БД (ODBC, JDBC и т.д.). Вторая – объектная библиотека (точнее библиотека классов), инкапсулирующая «черновую» работу с XDBC, и позволяющая приложению работать с БД в терминах объектов прикладной области и операций над ними. Библиотека zDAO играет очень важную роль: помимо того, что ее программный код составляет значительную долю от общего объема ППО (30%-40%), она задает для «образ мышления» приложений.

Название и идея zDAO заимствованы у одноименной библиотеки Data Access Objects, лежащей в основе продукта Microsoft Access™. MS DAO™ представляет БД в виде иерархически подчиненных объектов (напр. БД->Таблица->Поле), и обеспечивает сторонним продуктам, помимо доступа к БД разнообразных форматов, возможность манипулирования структурой БД формата MS Access™.

Именно эта идея, с необходимыми изменениями, положена в основу библиотеки zDAO – реализовать для каждой атомарной сущности класс, выстроить эти классы в иерархию подчиненности и обеспечить управляющие классы всеми необходимыми методами для управления жизненных циклом объектов, им подчиненных. На Рис. 10 показана схема такой библиотеки, обслуживающей ИС, имеющую систему разграничения доступа (SAM), предложенную на Рис. 4 (см. раздел 2.3.9 стр. 26), и некоторую гипотетическую прикладную область, представленную ранее в качестве примера квази-ER диаграммы (Рис. 5, стр 40).

Для понимания управленческих связей необходимо дать некоторые пояснения по объектам данных ИС, изображенной на квази-ER диаграмме:

- *Объект данных* Area представляет область ответственности, в которой находятся *объекты данных* типов Container и Document.
- ОД Container может *содержать* в себе другие *объекты данных* – Container и Document – находящиеся в той же Area (см. CEntry).
- ОД Document может *ссылаться* на другие объекты типа Document (см DRef).
- Пара таблиц DRow и DProperty является нормализованным эквивалентом для очень широкой таблицы DRow, с очень большим количеством полей, большая часть которых остается незаполненной. Поле DP.type – словарный код такого *поля-свойства*, а поля DP.nvalue и DP.tvalue – позволяют свойству иметь или числовое, или текстовое, или комбинированное значение.

- ОД `Area` имеет однозначное соответствие *области данных* (поле `AA.scope_id`), а принадлежность остальных ОД этой `Area` — определяется по их принадлежности *области данных*.

Главным управляющим объектом `zDAO` является `zSystem`, инкапсулирующий соединение с БД в терминах `XDBC` и представляющий всю ИС целиком. Он должен содержать методы для установления и завершения соединения с БД (`Login/Logout`) и методы для работы с объектами подчиненных классов. Для всех документо-подобных объектов (коими являются в т.ч. ОД) необходима реализация следующих методов: `Search/Open/Load/Save/Create/Delete`. Для прочих, например для объектов SAM, набор методов управления, обычно, повторяет логику управления нижележащими хранимыми процедурами *ядра системы*. Таким образом, вся функциональность универсального приложения может строиться вокруг одного экземпляра объекта `zSystem`.

С учетом семантики ОД `Area`, в предложенном примере `zDAO`, методы управления объектами классов `zdoContainer` и `zdoDocument` размещены в классе `zdoArea`, в предположении, что специализированное приложение, предназначенное для работы в области ответственности, будет строиться вокруг экземпляра объекта `zdoArea` также, как универсальное – вокруг `zSystem`. Деление методов управления объектами между различными классами носит характер идеологического посыла – в действительности объект zdoArea может просто инкапсулировать объект `zSystem`, который и будет выполнять реальную работу.

Поскольку количество объектов подчиненных другому объекту может быть так велико, что даже их индекс хранить в памяти приложения нецелесообразно, то в библиотеке `zDAO` отношения подчиненности разделены на две категории:

- *управляет* – означает, что управляющий объект содержит методы для манипулирования подчиненным объектом (создание, удаление, поиск, чтение, сохранение измененного состояния и т.п), но не содержит самих этих объектов, и не предлагает механизмов для их последовательного перебора. Например: объекта класса `zSystem` представляет собой всю ИС целиком, но не должен для этого загружать всю БД в память приложения.
- *содержит* – означает, что объект не только управляет подчиненными объектами, но, будучи получен определенным способом, содержит их все в локальной памяти приложения. Эта схема подчиненности применяется, например, для *объектов данных*, когда загрузка ОД приложением означает загрузку всех записей его составляющих и выстраивание их в порядке подчиненности.

Для классов `zDAO` можно выделить четыре парадигмы использования:

1. *класс-тип данных* – аналог составного типа данных в процедурных языках. Используется для хранения одной записи и ссылок на объекты-коллекции, содержащие подчиненные ей записи. Большая часть кода *классов-типов данных* порождается *кодогенератором*, по описаниям таблиц (подобным Таб. 18).
2. *класс-коллекция* – объектный аналог массива, предназначен для хранения набора объектов (точнее – ссылок на эти объекты). В наиболее чистом виде эта парадигма представлена в классах zList и zDictionary, объекты которых являются плоскими наборами однородных записей.
3. *класс-контейнер* – более сложный, чем *класс-коллекция*, набор ссылок на объекты, обычно состоящий из несколько коллекций (zList). Предназначен для описания сущностей более высокого порядка, чем обычный *класс-тип данных* (например: *объектов данных*).
4. *класс-суперконтейнер* – объекты таких классов управляют жизненных циклом неопределенного количества других объектов ИС. *Класс-суперконтейнер* содержит методы для взаимодействия с БД или другим внешним хранилищами информации. Для того, чтобы ОД получил свое отражение в приложении в форме объекта-контейнера, он должен быть загружен из БД методом Load *суперконтейнера* (например `zSystem.Load`), а для того, чтобы сделанные изменения были зафиксированы в БД, объект-контейнер должен быть сохранен методом `zSystem.Save`.

> **NB** В связи с тем, что фиксация изменений в БД не происходит одновременно с изменением содержимого объектов-контейнеров, в коллекциях из которых они строятся (прежде всего zList) необходимо предусмотреть механизм учета всех удаленных объектов-записей, чтобы в процессе выполнения операции «Save» произвести реальное удаление соответствующих записей в БД.

В классе `zSystem` необходимо реализовать несколько методов, повсеместное использование которых позволит оптимизировать работу приложений, не затрагивая их собственного кода:

- `getDictionary(dicname)` – возвращает объект класса zDictionary, содержащий словарь dicname. Словарь возвращается из локального кэша, и только если его тем нет – загружается из БД, после чего помещается в кэш.
- `getPS(sqlQuery)` – возвращает объект `Prepared Statement` библиотеки XDBC (или заменяющий его объект zDAO). В дальнейшем, здесь также можно реализовать кэширование, которое способно существенно повысить производительность операций с БД.

В методе getPS можно предусмотреть механизмы макроподстановок, выполняемых в зависимости от некоторых глобальных настроек объекта zSystem. Например для префиксов view: `{"SELECT * V%_SomeTable" -> { SELECT * V_SomeTable" или SELECT * VD_SomeTable"}}`, что позволит переключать приложение, написанное с повсеместным использованием этой нотации записи SQL запросов между использованием актуального состояния и среза истории. Естественно, при введении этой нотации все внутренние SQL запросы zDAO должны быть написаны с ее использованием.

В библиотеке zDAO следует придерживаться строгой статической типизации переменных и полей классов, что позволит снизить накладные расходы на внутренний контроль, переложив его на компилятор (или интерпретатор), и избежать проблем связанных с неявным преобразованием типов.

В некоторых случаях необходимо хранить и обрабатывать объекты в режиме нестрогой типизации значений полей. Достаточно востребованной операцией над иерархической структурой объектов, представляющей ОД или другую подобную структуру, является преобразование ее в некоторый более свободный формат, текстовый или близкий к текстовому, например: в документ XML, с возможностью обратного преобразования в *объект-контейнер* zDAO.

Если преобразование объектов zDAO в более свободный формат особых проблем не вызывает, то при обратном – возможно несоответствие данных необходимым критериям качества (например формату числа или даты). Обычно, после этого необходимо вернуть документ на доработку, с указанием причины отказа и места ошибки, причем формат возвращаемого документа может отличаться от формата исходного (типичный пример – интерактивное web-приложение, где документ может поступать в виде параметров CGI запроса, а возвращается на доработку – в форме html документа). В этом случае необходимо работать с данными после синтаксического разбора документа, в которых уже присутствует структура документа, но значения полей все еще представлены в том же текстовом виде, что и в исходном документе.

Для представления результатов синтаксического разбора такого документа, в библиотеке zDAO предусмотрена конструкция из двух классов – `zFieldSet` и `zField`. Объект класса `zField` – это одно именованное поле (`zField.name`), имеющее текстовое значение (`zField.value`). Объект `zFieldSet` – заменяет запись или объект-контейнер. Он содержит две коллекции: одну для объектов `zField`, другую для `zFieldSet`. С применением объектов этих двух классов можно закодировать любой *объект-контейнер* или *объект-тип данных*, со всеми его полями и содержащимися в нем объектами. Все классы `zDAO` должны содержать методы для преобразования своих объектов в структуру `zFieldSet/zField` и обратно, с порождением необходимой диагностики если преобразование невозможно. Если используемый язык не поддерживает строгой типизации, или требуется универсальный механизм обработки ОД заранее неизвестной структуры – возможно сделать `zFieldSet/zField` единственным способом хранения ОД в приложениях.

Насколько надежно объекты `zDAO` должны изолировать `XDBC` от остального приложения – вопрос дискуссионный. Скорее всего не следует ставить эту задачу в строгой форме, приложение может и должно пользоваться средствами `XDBC`, если соответствующих средств `zDAO` не предусмотрено или недостаточно. Действительное решение вопросов гарантированной изоляции приложения от всех внутренних механизмов лучше возложить на библиотеку «бизнес-объектов», когда (и если) у системы возникнет потребность в приложениях n-звенной архитектуры (Рис. 9).

В трех-звенной архитектуре, при дистрибуции функциональности *сервера приложений* через RPC/RMI/CORBA/EJB/DCOM или иной механизм, за основу этой функциональности можно взять

классы-суперконтейнеры, а для кодирования передаваемых *объектов-контейнеров* и *объектов-коллекций* – применять их преобразование в XML, или иной удобный формат. При этом, после соответствующей доработки, библиотека zDAO может использоваться приложением, взаимодействующим не с *ядром системы*, а *сервером приложений*. Вообще говоря, zDAO можно заранее спланировать как абстрактный слой, взаимодействующий с системой через различные zDAD (Data Access Driver), способные работать с различными СУБД или *серверами приложений*. В этом случае одно и тоже zDAO, но с разными zDAD будет работать и в приложении, взаимодействующим с сервером приложений, и на самом сервере приложений, работающем с СУБД. Впрочем, функциональность *серверов приложений* редко повторяет zDAO, но идея декомпозиция zDAO на предметный слой и слой, зависимый от типа источника информации, может оказаться продуктивной.

Последние несколько слов необходимо сказать о префиксах в именах классов (z,zr,zdr,zdo). Они являются разновидностью *венгерской нотации,* обычно применяемой к именам переменных для кодирования их реального или подразумеваемого типа данных. В данном случае она применена к именам классов. Использовать ли такой способ именования классов зависит от используемого языка программирования и личных убеждений проектировщика. В данном случае, префиксы на схеме Рис. 10 хорошо показывают назначение всех классов, а также позволят безболезненно иметь одновременно существующие классы zSystem и zdoSystem, zDictionary/zCode и zdoDictionary/zdrCode.

4.2. Правила взаимодействия частей приложения

Рис. 11 Базовые блоки прикладных приложений TIS.

Вплоть до программного кода приложений, реализующего пользовательский интерфейс или иной полезный прикладной процесс – как то обработку очередей заданий, автоматическую подготовку и публикацию отчетных документов, выдачу управляющих команд другим системам, репликацию областей данных и т.п. – взаимодействие частей системы можно представить как некоторый конвейер: ядро системы<->XDBC<->zDAO<->приложение. Внутри приложения связи могут быть гораздо более изощренными и запутанными, и если в них заранее не заложить некоторую систему – это может в дальнейшем существенно затруднить их доработку и развитие.

В самой простой ИС, выполненной с применением TIS подхода, будет минимум два приложения. Первое – предназначено в основном администратору системы и служит для управления объектами СРД и прочим общесистемными задачами. Второе – предназначено для обычных пользователей и обслуживает *объекты данных* прикладной области.

На Рис. 11 представлены конструктивные блоки двух таких гипотетических приложений, реализующих документо-подобный пользовательский интерфейс к библиотеке zDAO, предложенной ранее на Рис. 10. Каждый прямоугольник представляет собой комплекс программного кода, обрабатывающий некоторую документо-подобную сущность, а все эти комплексы подобны друг-другу как по внутреннему устройству, так и по внешнему программному интерфейсу.

Рекомендуемая стратегия разработки предлагает придерживаться для этих комплексов общих шаблонов проектирования, стандартизовать для них общий программных интерфейсов обращения извне, а также интерфейс, по которому подобный комплекс взаимодействует с приложением. Это

позволит достаточно легко обращаться из одного документа к другому, не заботясь о деталях внутренней реализации последнего, и строить композитные документы, когда один документ должен полностью или частично отображаться в контексте другого. Следует стремиться к вытеснению всего общего кода в отдельно-сопровождаемые модули, а от каждого комплекса модулей, обслуживающего тип документа, необходимо добиваться самостоятельности мини-приложения (подобного апплету), которое будучи обеспечено необходимой «жизненной средой», может быть «пересажено» в любое другое приложение.

Таким образом, прямоугольные сущности на Рис. 11 – это модули-кирпичи из которых могут строиться любые приложения, а обозначенная на них функциональность – это минимальный набор операций над документами этих типов, которые должны быть доступны пользователю, и одновременно – ожидаемый набор методов программного интерфейса, доступного другим модулями. Ориентировочный размер такого модуля – 1-2 тыс. строк – позволяет рассматривать его как отдельное конечное задание, выдаваемое одному программисту на 1-2 недели.

5. Управление разработкой ИС. Прогнозирование и планирование.

Граничные условия и ключевые рекомендации к разработке ИС – памятка участника проекта
1. ИС должна обладать свойствами, достаточными для решения задач, ради которых она создана.
2. ИС должна обладать свойствами, необходимыми для ее внедрения и последующей эксплуатации в конкретных условиях, конкретного потребителя.
3. *Задача планирования*. Перед началом работ необходимо:
3.1. определить технические, человеческие, временные и финансовые ресурсы, необходимые для проектирования и разработки системы (в т.ч. для написания документации). Определить объемы программного кода ИС, состав и объем документации, сроки и бюджет разработки.
3.2. установить контрольные точки, в которых определяется соблюдение сроков и бюджета.
3.3. определить технические, человеческие, временные и финансовые ресурсы, необходимые для тестирования, сертификации и внедрения системы. Определить сроки и бюджет тестирования, сертификации и внедрения по отдельности.
3.4. определить сроки и бюджет на обучение сотрудников новой системе.
3.5. определить сроки и бюджет на доработку системы в процессе сертификации и внедрения.
3.6. определить ежегодный бюджет на сопровождение (и развитие) системы
4. *Задача реализации*. В процессе разработки необходимо:
4.1. На регулярной основе (предпочтительно еженедельно, в пятницу, в конце дня) проводить совместное совещание всего коллектива разработчиков, на котором подводить итоги сделанному каждым участником (и всеми вместе), определять и перераспределять индивидуальные задачи, обсуждать возникающие проблемы. Все достигнутые соглашения должны фиксироваться письменно.
4.2. На ежедневной основе производить количественные замеры сделанного каждым участником проекта. Количественный замер должен сопровождаться кратким описанием того качества, которое привнесено в проект измеренным количеством. Эффективнее всего если каждый участник будет вести свой дневник разработки, куда ежедневно делать короткую запись (3-5 строк) с количественным и качественным описанием сделанного.
4.3. Руководитель проекта должен регулярно анализировать состояние процесса разработки, удачное и неудачное прохождение контрольных точек. По результатам анализа – вносить коррективы в процесс, переопределять и перераспределять индивидуальные задачи. Результаты анализа и принятые решения должны фиксироваться письменно.
5. Система должна быть разработана в пределах заданных сроков и бюджета.
6. Система должна быть оттестирована и сертифицирована в пределах заданных сроков и бюджета.
7. *Задача внедрения*. Системы должна быть внедрена, в ней должны быть сделаны необходимые доработки, в т.ч. связанные с новыми потребностями, возникшими, пока велась разработка.

Таб. 21 Памятка участника проекта

Последняя содержательная глава этой работы посвящена самому сложному аспекту разработки ИС – тому, с чего она должна начинаться и чем руководствоваться на всей своей протяженности. Я, на протяжении всего предшествовавшего изложения, избегал применения личного местоимения, с

которого начинается это предложение, и тому были свои причины – исключение личного аспекта делает изложение менее дискуссионным и, возможно, более конструктивным. Но гипотезы и предложения, содержащиеся в этой главе, могут вызвать серьезное отторжение и ожесточенную дискуссию, прежде всего по причине существенного примитивизма. Пусть эта критика будет персонифицированной и отдельной от критики самого TIS/SQL подхода, также как излагаемые здесь идеи существуют сами по-себе и распространяются мною на любую разработку.

Отрасль знания, соответствующая теме этой главы, называется «Инженерия разработки ПО» (Sofware Engeniring), а продуктом ее являются разнообразные «методологии разработки ПО», т.е. *учения* о том, как правильно и в срок разрабатывать программные продукты, при этом, по возможности, не превышая бюджета. Первая проблема состоит в том, что слышали об этом многие, а вот реально используют – единичные коллективы разработчиков. Вторая проблема, частично объясняющая первую, состоит в том, что любая методология может функционировать только в коллективе, являясь его «коллективным знанием» и существенным образом определяя профессиональные отношения в нем. Таким образом, даже если отбросить проблему выбора конкретной *методологии* из многих, ее внедрение в существующем коллективе будет затратным и конфликтным. Альтернативный путь – построение коллектива вокруг выбранной методологии – фактически построение нового предприятия, со всеми сопутствующими расходами. Ко всему сказанному есть еще и третья проблема: никакая методология не является гарантией успеха конкретного единичного проекта, единственное на что она действительно способна – это сделать процесс разработки более предсказуемым и поддающимся последующему анализу, но только при условии систематического и осмысленного применения.

В реальности, разработкой большей части ИС занимаются стихийно сложившиеся коллективы, которые в случае ее внедрения, что называется «при ней живут, и при ней же кормятся», иногда, под давлением руководства, пытаясь разрабатывать еще что-то. Очень часто в начале разработки такой коллектив не может определить ни сроков разработки, ни ее объемов, а зачастую не имеет даже достоверного технического задания. В этом случае о применении какой-либо методологии говорить не приходится (впрочем «Экстремальное программирование» предлагает использовать себя и в таком «экстремальном» случае, но для этого надо быть к нему готовым).

Здесь я хочу предложить очень простой, в стиле древнегреческих геометров (т.е. «с помощью палки и веревки»), способ оценить предполагаемый объем программного продукта и необходимых трудозатрат на его создание, а также дать рекомендации по сокращению календарного времени, в течении которого он будет разрабатываться.

В Таб. 21 представлена полу-страничная памятка, которой можно воспользоваться, начиная разработку ИС, если у вас нет аналогичной, составленной в соответствии с принятой вами методологией (если же она есть, то их полезно сверить). Главным правилом, которым я рекомендую руководствоваться при решении всех промежуточных задач и проблем, выбрано «разработать в срок и соблюдением бюджета». При этом сроки и бюджет должны быть ограничена, поскольку задача без этих ограничений скорее всего будет решаться вечно.

Под понятием «разработано», здесь и далее, будем понимать полнофункциональную бета-версию системы, которую можно передать заказчику (потребителю) для оценки соответствия его потребностям и постановки в тестовую эксплуатацию. Это не совсем классическое определение, фактически на этой стадии выполнена только половина работы по созданию системы, как по трудо-временным затратам, так и по бюджету. Однако это очень важный рубеж, после которого программисты перестают быть главной движущей силой проекта, а его дальнейшее движение зависит от тестировщиков, сертифицирующих органов, внедренцев и т.п. Этот момент похож на высшую точку восхождения на горную вершину, в которой цель кажется достигнутой, но после которой необходимо спуститься вниз, фактически проделав точно такой же путь, как и для восхождения. Эта аналогия тем более уместна, что в обоих ситуациях риск катастрофы существенно возрастает после прохождения этой «высшей» точки.

По этой причине, в «памятке», я предлагаю четко разделять бюджеты «на разработку» и «на тестирование, сертификацию и внедрение». Первый более или менее можно спрогнозировать в процессе составления проекта ИС (или даже сразу – из технического здания), а остальное – сильно зависит от тех условий, в которых система будет внедряться и эксплуатироваться.

Прежде чем переходить к оценке и планированию разработки, следует коротко рассказать о той работе, которая последует за «разработкой», и чей финал обозначено в памятке как «Задача внедрения». Последнее время, оформился целый класс программных продуктов, которые начинают эксплуатироваться с состояния частично-работоспособной бета-версии и доводятся в процессе эксплуатации, причем их дальнейшее развитие, которое может длиться годами, принципиально не отличаясь от доводки. В этом случае вполне правомерно совместить процессы тестирования и внедрения, а заодно и их бюджеты. В ином случае бюджет тестирования (и, сопоставимый с ним, бюджет сертификации, если она необходима) может лежать в диапазоне от 10% до 100% бюджета на проектирование и разработку. Если же любой из них превышает 100% бюджета разработки, то это повод насторожиться и разобраться в причинах подобного дисбаланса.

Календарное время, необходимое для полномасштабного тестирования программного продукта – сопоставимо с календарным временем, потраченным на его разработку. Моя оценка – 25%-50%, причем тестирование ни при каких условия не может занять меньше календарного месяца (а скорее даже двух-трех). Совместимость тестирования и сертификации в одном процессе и одном бюджете – маловероятна, равно как и их параллельное выполнение. Если продукт требует сертификации, то полномасштабное тестирование должно ему предшествовать. Тем не менее, внедрение и обучение пользователей можно начинать параллельно тестированию/сертификации.

После прохождения «высшей» точки разработки можно задуматься о сокращении или расширении команды разработчиков, которое допустимо осуществить после прохождения первых стадий тестирования. В случае сокращения – я не рекомендовал бы делать ее меньше двух-трех человек, а в случае увеличения – целесообразно «пропустить» всех потенциальных программистов через команду тестирования системы. Увеличение команды целесообразно если планируется интенсивный рост системы после внедрения, а также в случае прогнозируемого ухода из проекта одних разработчиков, и замены их другими, которые будут заниматься сопровождением системы на постоянной и долговременной основе.

Чем дальше проект отходит от своей «высшей» точки, тем большее значение принимает поиск компромиссов и диалог между внедренцами, пользователями и программистами, а также умение последних решать возникающие проблемы с минимальными воздействием на систему.

На этом хочу закончить короткий реверанс в сторону проблем тестирования, сертификации, внедрения, обучения пользователей и т.п. Очевидно, что по совокупности все это потребует сил, средств и времени не меньших, чем проектирование и разработка, планированию и прогнозированию которых посвящено дальнейшее повествование.

Решение «задача планирования» базируется на правильной оценке количества и качества работ, необходимых для проектирования и реализации системы. Результатом этой оценки будет время, теоретически необходимое на разработку системы одним разработчиком. Далее эту работу надо эффективным образом разделить на изолированные части и распределить между всеми участниками проекта, чтобы привести календарное время разработки к разумному.

При оценке оставим за скобками составление технического задания и выбор конкретных программных и аппаратных средств, в т.ч. и конкретной СУБД, с оценкой ее соответствия потребностям TIS/SQL подхода. Время, затрачиваемое на эти задачи, начинается от одного месяца и заканчивается бесконечность, для проектов не дошедших до следующих стадий. (В процессе адаптации TIS/SQL подхода к новой СУБД, целесообразно начать с реализации ее средствами «простейшей TIS» (Рис. 2 на стр. 9), исходный код которой представлен в приложении В.)

Предположим, что у нас есть техническое задание или черновой вариант проекта, по которому можно определить количество *типов объектов данных* в создаваемой системе. В этой ситуации можно сделать прогнозную оценку необходимых работ по аналогии с существующим программным продуктом, выполненном по сходной технологии.

В качестве референсной системы, в Таб. 22, приведены количественные данные по слегка сокращенной и доработанной системе YELL, разработанной мной для РосНИИРОС (http://ripn.net). Главные ее отличия от TIS/SQL – отсутствие мандатной СРД, журнала аудита, состояния транзита, минимум view (только V_ и VH_), другие имена системных таблиц и полей.

Подсистема	таблицы		число полей	view		процедур		кол-во типов ОД или док-тов	тесты	zDAO	код прилож. [т. строк]	всего [тыс. строк]
	[шт]	[тыс. строк]		[шт]	[тыс. строк]	[шт]	[тыс. строк]		[тыс. строк]	[тыс. строк]		
Словарь	**7**	1,4	*28*	*7*	0,2	*11*	0,4	**-**	0,1	0,4	-	2,5
SAM	**10**	0,1	*49*	*10*	0,2	*56*	3,0	**3**	0,3	0,6	2,5	6,7
Прикладная обл.	**16**	0,4	*116*	*32*	1,5	*109*	10,0	**5**	-	4,5	8,0	24,4
Общий код	**3**	0,10	*17*	*2*	0,1	27	1,4	-	1,7	2,6	6,5	12,2
make-файлы												2,0
Кодогенератор												1,6

Таб. 22 Оценка объемов разных частей реальной референсной ИС (5 ОД, 36 таблиц, 50 тыс. строк)

Референсная ИС выполнена с применением СУБД ORACLE, *ядро системы* написано на языке PL/SQL, а прикладная часть выполнена в виде web-приложений, написанных на Java™ и функционирующих под управлением сервера Apache Tomcat (http://tomcat.apache.org/). Java используется только на сервере, клиент использует любой web-браузер (без Java). В представленной системе 5 типов *документов* (ОД), 2 *области данных* без передачи ОД между ними, сложная система словарей, задающих допустимые формы для двух типов ОД. Настройки СРД статичны (подобны словарю). Число строк для таблиц словаря учитывает не только скрипты для их создания, но и скрипты для заполнения (т.е. дает представление о количестве кодов в них). Главная задача системы – накопление и отображение информации, с учетом выявляемых связей между документами.

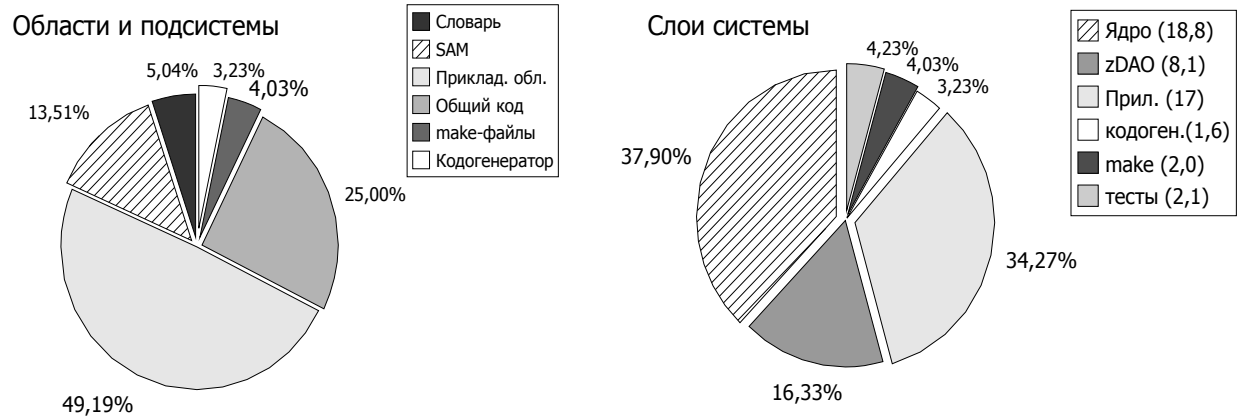


Рис. 12 Распределение кода по областям, подсистемам и слоям референсной ИС

Диаграммы на Рис. 12 и Рис. 13 построены по данным Таб. 22, и дают наглядное представление о распределении программного кода с различных точек зрения («разрезов»). Каждая диаграмма на Рис. 12 учитывает 100% кода всей системы, а на Рис. 13 - каждая использует одну строку Таб. 22. Данные о количестве типов ОД (или таблиц) могут использоваться для прогнозирования объемов кода и количества прочих сущностей (процедур, view и т.п.) в аналогичных областях разрабатываемой ИС (прежде всего в «прикладной области», где происходит основной рост системы). Следует иметь в виду, что при разработке полноценного оконного приложения, объем его кода будет больше на 20%-30%, чем у соотв. web-приложения в Таб. 22 (оценка по данным ИС KI-MACS).

Рис. 13 Распределение кода в приложениях

Для оценки объемов кода ИС используется довольно спорная и субъективная единица измерения [тысяча строк программного кода]. Еще более спорная оценка будет предложена для рекомендуемой «крейсерской» скорости написания кода одним программистом – 300 строк кода в день, что совпадает с декларируемой предельной производительностью писателя. Далее привожу систему гипотез, своеобразное credo, на основе которой сделаны все дальнейшие расчеты и рекомендации.

Гипотеза о строке программного кода. Чтобы не растекаться мысию по древу в спорах о недостатках и неопределенностях понятия «строка кода», хочу отметить первое – это самый доступный способ измерения; второе (и главное) – строка кода это некая синтаксически оформленная мысль, сходная с иероглифом, и воспринимаемая целиком, вплоть до того, что пустая строка (коих в приводимых данных от 2.5% до 6.5%), поставленная в определенном месте, также несет свой беззвучный смысл. Не говоря уже о строках комментариев, включаемых во все измерения безоговорочно.

Гипотеза о длине хорошо мотивированного дюйма. В течении четырех тысяч лет документированной истории (и десятков тысяч, им предшествовавших) человечество использовало единицы измерения длины привязанные к геометрически характеристикам человеческого тела (дюймы, футы, ярды, локти, шаги и т.п.). Много неудобств и забавных казусов связано с тем, что все люди разные, и измерения произведенные по одному человеку, будет несколько отличаться от измерений по другому. И тем не менее человеческое тело это то, что у него всегда при себе, а вероятность того, что где-то дюйм окажется равен ярду – чрезвычайно мала. Иными словами, вся цивилизация построена с применением неточных средств измерений и субъективных эталонов, связанных с реальным человеком, в предположении, что эти эталоны можно экстраполировать на все остальное человечество. Я предлагаю распространить это предположение и на измерения интеллектуально/ремесленной деятельности, по крайней мере среди людей сопоставимого интеллектуального, образовательного и профессионального уровня: *Пусть некий человек A, выполнил работу B за время T, получив работоспособное решение объемом N, тогда некоторый другой человек A1 (или тот же A), будучи хорошо мотивирован на достижение результата, и имея известные показатели человека A, взявшись за задачу B1, аналогичную B, способен решить ее за время T1, близкое к T, и уложить это решение в объем N1, близкий к N.*

Гипотеза 10-ти строк. Считается, что после завершения всех стадий разработки, тестирования и написания документации, средняя производительность одного участника проекта, включая непрограммистов, составляет 10-15 строк программного кода в день.

Гипотеза 300-т строк или 5-ти страниц (*Гипотеза о предельной суточной производительности писателя*). Программист (или писатель) способен произвести в течения суток ограниченный объем мысли, выраженной в программном коде или тексте. Предельный объем суточной производительности, принятый в этой работе составляет около 300 строк кода, или 3-5 страниц текста формата A4 (прибл. 4500 знаков [с пробелами] на страницу). Объем кода не зависит от используемого языка программирования. Личные ограничения могут варьироваться, иногда существенно, но я предполагаю, что вариации связаны более с формой записи, избыточной или компактной, чем с самим количеством реализованной мысли. В любом случае, только ежедневные замеры производительности конкретного программиста, делают возможным прогнозирование и планирование его работы.

Гипотеза калибровки объема кода по эталону. Сравнивать производительность разных программистов, выраженную в строках в день, – не совсем корректно, но часто необходимо для прогнозирования. В этом случае можно прибегнуть к калибровке их индивидуальных измерений по эталону – взять результаты замеров, полученные при выполнении первым и вторым программистом двух эталонных (равнозначных или одинаковые) работ, после чего сравнить объемы программного кода(N1э,N2э) и потраченное время(T1э,T2э): `{T1(T2)=T1э/T2э*T2; N1(N2)=N1э/N2э*N2}`.

Гипотеза о лучших и худших программистах. Есть мнение, что производительность лучших и худших программистов различается в 10-ть раз. Считаю, что основное различие между ними состоит не в различной предельной производительности, а в способности (умении) выстраивать процесс своей работы (мышления) так, чтобы на протяжении длительного периода получать ежедневную производительность близкую к предельной. Там не менее разница есть – ее следует учитывать.

Гипотеза о предельной суточной производительности читателя. В процессе разработки приходится не только писать, но и читать. (Следует вспомнить, что программисты должны те только писать код, но и, как минимум, читать технические спецификации и код других программистов). Скорость чтения очень существенно зависит от литературных достоинств текста. У меня есть три

книги, объемом 413, 564 и 284 страницы (формат A5≈1/2 от A4), которые по утверждению «экспертов» были прочитаны ими за один день. Таким образом, если усреднить и перевести этот объем в формат A4, получится: (413+564+284)/3/2=210 страниц в день. По-видимому, это и есть предельная дневная производительность читателя, однако технические тексты и исходные коды программ не обладают выдающимися литературными качествами, поэтому полагаю предельную производительность их читателя на уровне 3 тысяч строк кода или 30-50 страниц документации в день.

Гипотеза взаимосвязи предельной производительности при чтении и письме. Полагаю, что между предельной производительность человека как писателя и как читателя, есть связь, а 5 страниц написанного текста, есть результат скрытого, внутреннего «проживания» 200 страниц размышлений.

Что касается средней производительности программиста, то при работе в коллективе она обычно составляет 30%-60% от предельной. Остальное время тратится на общение (в коллективе и с заказчиком), планирование, согласование, и прочие производственные и непроизводственные задачи. Так в одном проекте, на стадии кодирования, почти все сделанные мною личные дневные замеры показали производительность 270-300 строк в день, а средняя производительность за исследованный календарный период составила 170 строк в день. Объяснение просто – в остальные дни моя работа не была связана с написанием кода, и эти дни в учетные записи просто не попали. По этой причине я рекомендую делать записи в индивидуальный журнал учета на каждый календарный день, чтобы было легче потом определять незапланированные задачи, на которые уходит рабочее время.

В любом случае, каждый программист в проекте должен стремиться, чтобы его ежедневная производительность, на достаточно протяженных участках времени, была близка к предельной. Для этого необходим четкий план работ (прежде всего внутренний, личный), когда завершая рабочий день есть четкое представление, чем заняться с утра, и очевидный фронт работ и на завтра, и на послезавтра. (Например: в конце дня, я часто начинаю работу, запланированную на завтра, чтобы было легче втянуться в нее утром). Помимо этого, программист должен быть мотивирован на выполнение работы в указанные сроки и, что немаловажно, – осознавать их реальность и обоснованность.

Наверное предельная производительность достижима ежедневно, если над проектом работает только один, но сильно мотивированный разработчик, который не нуждается в согласованиях и способен держать весь план разработки в голове. Однако такой программный продукт оказывается очень сильно связан с этим единственным создателем и может просто не поддаваться отчуждению и последующему развитию кем-то другим.

При работе над проектом команды из двух хороших разработчиков, можно ожидать, что 30%-50% своего времени они потратят на переговоры и согласования, при этом напишут несколько больше кода, чем надо, и в результате выполнят работу за тот же календарный срок, что и один сильно мотивированный одиночка из предыдущего варианта (правда, если при этом не перессорятся, и завершать проект не придется кому-то одному). Скорее всего результат такой разработки будет более стабилен, предсказуем и проще в дальнейшем сопровождении и развитии.

Для большей стабилизации команды в нее стоит добавить третьего участника, возможно не программиста, а скажем технического писателя, тестировщика и читателя программного кода в одном лице. Объем необходимой документации я обычно оцениваю как 10%-20% от объема программного кода продукта, но помимо него есть внутренняя переписка, которую нужно обрабатывать и систематизировать, так что такому специалисту всегда будет чем заняться.

И наконец, к этой команде из трех человек, надо добавить четвертого – руководителя проекта, который будет заниматься планированием, координацией работ, отвечать перед заказчиком, добиваться необходимых услуг и ресурсов от сторонних поставщиков, улаживать конфликты и т.п.

Таким образом получается команда разработчиков из четырех человек. В ней отсутствуют системный архитектор (человек спроектировавший ИС), системный администратор (тот, кто настраивает компьютеры, ПО, сети и периферию), администратор БД (человек, хорошо разбирающийся в конкретной СУБД и отвечающий за ее настройку и функционирование). Будем исходить из предположения, что в небольшом коллективе все или некоторые члены – специалисты в нескольких областях, и выполняют сразу несколько функций. Иначе всех этих отсутствующих специалистов надо включать в команду или привлекать со стороны, по мере необходимости.

А теперь можно сделать ориентировочный расчет как теоретические 300 строк в день превращаются в 10 реальных. Есть 4 участника; из них 2 – программисты с производительностью 150

строк в день (50% от 300); после достижения «высшей» точки разработки практически весь код уже написан, но необходимо потратить еще столько же времени на тестирование и доводку (т.е. на каждый день кодирования – один день комплексной доработки, т.е. делим производительность еще на 2): `S=150*2/4/2=37.5[строк/день];` плюс сделайте поправку на мой оптимизм, или поищите неучтенных участников (например команду тестировщиков) и незапланированные работы (разработку проекта, настройку рабочей среды, изучение новых средств и программных продуктов).

Возвращаясь к Таб. 22, попробуем сделать несколько численных оценок

1. рассчитаем объем кода (в тыс. строк) некой TIS, представленной на квази-ER диаграмме (Рис. 5):
   1.1. В словаре референсной системы 7 таблиц и 2.5 [т.строк], в то время как в нашей TIS мы можем обойтись одной таблицей, тогда объем словаря: 2,5/7=**0,357** [т.строк]
   1.2. Подсистемы SAM почти одинаковы – примем данные без изменений: **6.7** [т.строк]
   1.3. В прикладной области референсной системы 5 типов ОД и 24,4 [т.строк], т.е. 24,4/5=4,88 [т.стр/тип ОД]; а в TIS системе – 3 ОД, что дает 4,88*3 = **14,64** [т.строк]
      1.3.1. некоторые поправки нужно внести объем кода ядра, необходимый для реализации одного ОД: для референсной системы это (24,4-(4,5+8,0))/5=2,38; однако TIS несколько сложнее, по схеме ядра (Рис. 7), в ней на каждую таблицу приходится на 4 view (6 вместо 2) и на 2 процедуры больше, кроме того на каждый ОД приходится еще 7 дополнительных процедур. В среднем, в каждом ОД 3 таблицы, что дает добавочный код для 4*3=12 view и 7+2*3=13 процедур. В среднем, одно view это 1,5/32≈0,05; а одна процедура это 10/109≈0,09 [т.строк]. Тогда добавочный кода для одного ОД составит 12*0,05+13*0,09≈**1,8** [т.строк].
      1.3.2. Таким образом добавочный код для 3-х ОД составит 3*1,8=**5,4** [т.строк]
   1.4. «Общий код» – примем без изменений: **12,2** [т.строк]
   1.5. Инфраструктуру make-файлов, также примем без изменений: **2,0** [т.строк]
   1.6. кодогенератор – без изменений: **1,6** [т.строк]
   1.7. Итого программного кода: **0,357+ 6,7+14,64+5,4 +12,2+2,0+1,6≈42,9** [т.строк]
   1.8. Минимальный объем документации – это 10% от объема кода: **42,9*0,1≈4,3** [т.строк]
   1.9. Итого всего: **42,9+4,3=47,2** [т.строк] (без учета пользовательской документации)
2. Рассчитаем трудозатраты, для написания такой системы (до стадии работоспособной бета):
   2.1. теоретический минимум, при 300 [строк/день], 100% КПД и 22 рабочих днях в месяц: 47,2/0,3/22≈7,2 человека-месяца. Этот результат практически недостижим, но важен для осознания масштаба затрат. Кроме того – это календарный срок за который систему реально может разработать слаженная команда из двух программистов и некоторого количества управляющего и вспомогательного персонала. Если этот срок удвоить: 7,2*2=14,4 [календарных месяцев], то получится вполне реальный срок от начала разработки до состоявшегося применения системы, при «экстремальных» способах разработки и внедрения.
   2.2. Более реальный минимум трудозатрат – это удвоенный или утроенный теоретический, т.е. 7,2*2=14,4 или 7,2*3=21,6 [человеко-месяцев] на разработку, однако он не учитывает работу непрограммистов, что может удвоить и эти цифры. Впрочем, нельзя недооценивать важность всех участников команды, даже если от них не зависит написание кода и документации, от них может зависеть то, что это все вообще будет сделано, и при этом соответствовать потребностям конечных пользователей.

В действительности, руководитель проекта и технический писатель могут одновременно работать в нескольких проектах, а написание разных частей системы может потребовать привлечения разных специалистов, не говоря уже о команде тестирования, которую нужно задействовать эпизодически, на конкретных стадиях. Оценка работ в человеко-месяцах, это в большей степени жонглирование цифрами перед бухгалтерией, для достижения компромиссного бюджета. В этом разделе дано достаточно пищи для подобных спекуляций, позволяющих оценить «массу» проекта, но далее эта «масса» должна быть согласована с имеющемся «двигателем» – командой разработчиков.

NB Разработка сложного ПО – это прежде всего командная игра. Причем, зачастую не одной команды, а нескольких, находящихся в определенной конфронтации друг с другом: проектировщики, программисты, тестировщики, внедренцы, пользователи. Считать трудозатраты в человеко-месяцах можно только применительно к конкретному участнику внутри команды и конкретной работе, за которую он отвечает от начала и до конца. Оценивать весь проект следует в командо-месяцах реальных команд, работающих в реальных условиях, над конкретными целями. А к правдоподобным теоретическим оценкам этих сроков, необходимо добавить важнейший ингредиент: обязательство каждого члена команды сделать все от него зависящее, для достижения общей цели в оговоренный срок.

5.1. Распараллеливание и стабилизация разработки

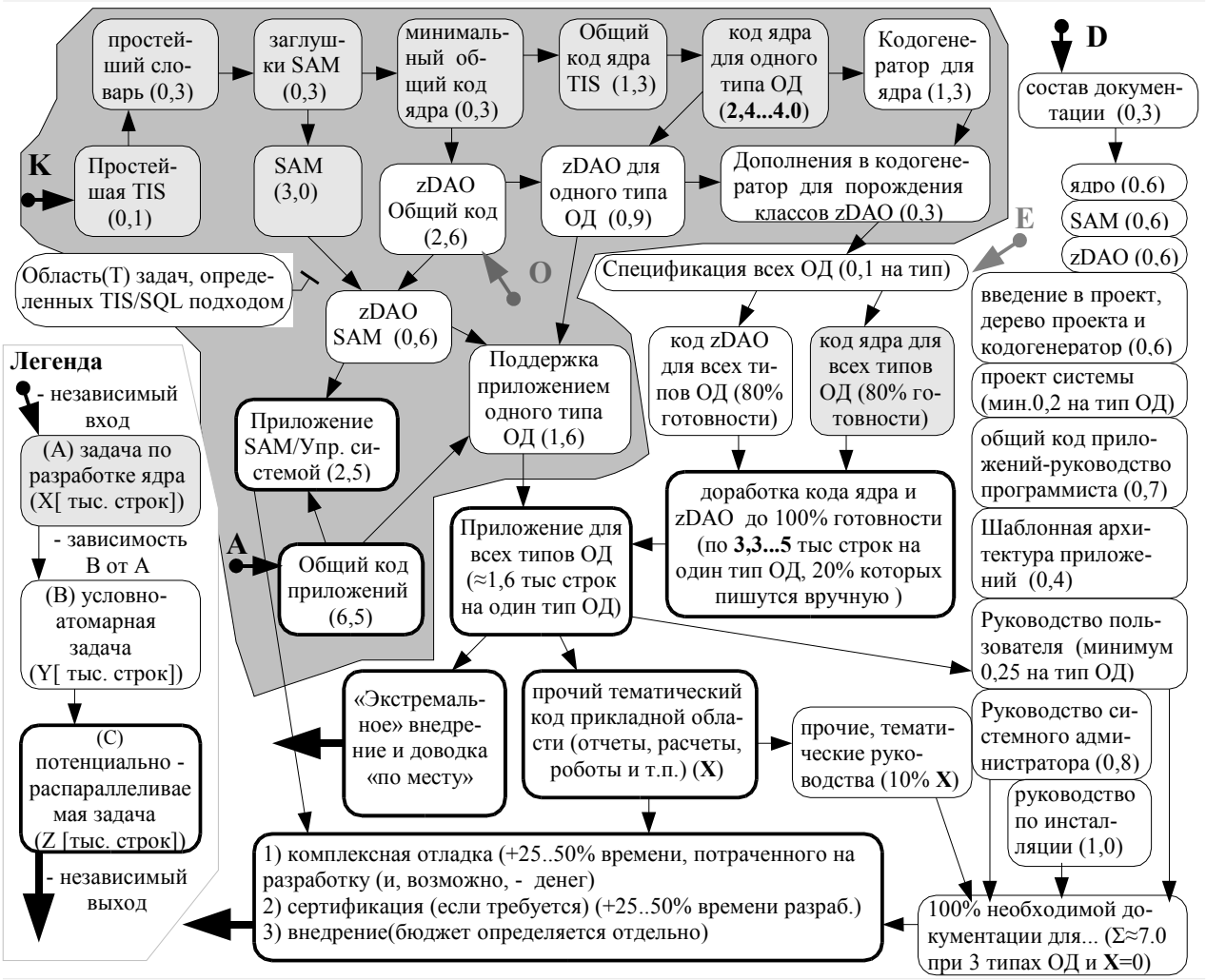


Рис. 14 Сетевой график разработки TIS

При работе над проектом нескольких программистов, встает вопрос эффективного распределения работы между ними, что требует деления ее на отдельные задачи, установления зависимостей между ними и назначения конкретных исполнителей (т.е. «*распараллеливания*», здесь и далее этот термин имеет именно такое значение – о терминах не спорят, о них договариваются). Одним из методов подобного планирования является «сетевой график работ», эталонный вариант которого, для разработки типовой TIS с «нулевой» стадии, представлен на Рис. 14. Исходные данные взяты в основном из Таб. 22, с некоторыми поправками, полученными выше, в процессе расчета объемов кода TIS, по квази-ER диаграмме (Рис. 5).

Почти для каждой задачи дан ее абсолютный или относительный объем в тысячах строк кода. Однако, приведенные цифры не учитывают тесты и инфраструктуру make-файлов, что суммарно составляет 8%-10% от общего кода системы (см. диаграмму Рис. 12), и этот объем должен быть учтен в общей смете на систему, чтобы не занижать действительного объема работ. Дилемма состоит в том, что эти дополнительные объемы не составляют отдельных задач, а получаются как побочный продукт разработки, по мере возникновения в них потребности, со временем эволюционируя в некоторую самостоятельно значимую инфраструктуру (например: набор включаемых make-файлов). Вписывать в каждую задачу, по 8-10% сразу – значит затемнять ее истинный вес. Возможно что время, необходимое на выполнение этих работ, лучше всего резервировать по 2-3 дня после каждой контрольной точкой, когда выполнен значительный объем работ, и всем нужно остановиться, осмотреться и подумать. Чтобы программисты в эти несколько дней занялись упорядочением и систематизацией написанного кода.

В TIS имеется два четко разделенных пространства (слоя) разработки – *ядро системы* и все остальное, условно именуемое «прикладное ПО». Граница между ними не только формальная, но и языковая: ядро пишется на обычном и процедурном SQL, а все прочее – на универсальных языках. Такое положение делает очевидным возможность и желательность наличия в команде двух программистов, отвечающих каждый за свой слой. Диаграмма на Рис. 12 показывает сопоставимость количества кода в обоих пространствах (по крайней мере в начале жизни ИС), а если пренебречь zDAO – то они практически равны. В таком тандеме архитектор проекта работает над ядром, кодогенератором, проектной документацией, выражением предметной области ИС в терминах *объектов данных*, курирует архитектуру и функциональность zDAO. Второй разработчик – отвечает за архитектуру прикладных приложений и пользовательский интерфейс. Кто из двух пишет код zDAO – вопрос дискуссионный, возможно, что здесь есть место третьему программисту, который в равной степени ориентируется в обоих пространствах и языках. Такой человек, будучи «полиглотом», способен «подстраховать» весь проект, беря на себя задачи от каждого из двух архитекторов, разгрузив их для лучшей проработки дизайна и написания рабочей документации. Возможно, что этот третий и должен быть главным архитектором и идеологом проекта, отвечая за проектную документацию, состав и структуру типов ОД, кодогенератор и концептуальную целостность всей системы, а программирование для него – хобби, «чтобы шпага в ножнах не ржавела». В любом случае, пора вернуться к Рис. 14, чтобы разобраться как несколько человек смогут одновременно работать над одной ИС, не мешая друг-другу и теряя времени попусту.

На сетевом графике обозначено три очевидных точки независимого входа в проект – K,A,D; две условно-независимого – O,E; и две точки выхода из проекта, соответствующие двум вариантам развития событий после достижения «высшей» точки разработки – «экстремальному» или «классическому» тестированию и внедрению. Каждая точка независимого входа – это возможность с самого начала задействовать отдельного программиста (или даже группу программистов). Рассмотрим эти места одновременного вступления в проект с точки зрения, «какое максимальное число программистов можно задействовать»:

- K – «ядро». В этом месте можно войти группой из трех программистов, во главе с *архитектором системы*, который сначала пишет и отлаживает простейшую TIS, потом объясняет двум коллегам как это должно работать и предлагает им 3-4 дня поэкспериментировать с этой «моделью на выброс», разработать для нее некоторую дополнительную функциональность и поискать уязвимости в используемых приемах. За это время он создает «словарь», «заглушки SAM» и «минимальный общий код ядра», после чего устраивает общее совещание, на котором происходит обмен мнениями и достигнутыми результатами, что завершается делением работ между двумя оставшимися программистами: первый (назовем его *безопасник*) начинает заниматься реализацией «SAM», а второй (*прикладник*) – «общего кода ядра TIS» и «кода ядра для одного типа ОД».

  В это время *архитектор* переключается на «zDAO, общий код», двигаясь в сторону «кода zDAO для одного типа ОД», а от него – к проверке, тестированию и анализу сделанного *прикладником*. За этим следует написание кодогенератора, после чего возникает условно-независимая точка входа «E», в которой весь код эталонного типа ОД подвергается перестройке кодогенератором, в процессе которой происходит доводка последнего. После этого открывается возможность

массовой кодогенерации и распараллеливания работы на основе спецификаций типов ОД, работа над которыми может происходить достаточно независимо друг от друга. С этого момента *архитектора системы* может сосредоточится на работе со спецификациями типов ОД и написании кода ядра и zDAO, требующего ручного кодирования.

В это же время, *безопасник*, завершив SAM, проводит предварительное тестирование корректности ее использования остальными частями ядра, письменно фиксирует и передает свои замечания остальным членам группы, после чего переходит к реализации «SAM zDAO», а в заключение – подключается к разработке «Приложение SAM/Управление системой».

Прикладник может уделить 2-3 дня чтению кода SAM и публикации своих замечаний о нем, после чего подключиться к совместной работе с *архитектором*, над кодом ядра, порожденным кодогенератором для новых типов ОД.

- А – «приложения». Здесь все несколько неопределеннее, задача «общий код приложений» обозначена как «потенциально распараллеливаемая», однако каким образом это можно сделать на графике не видно. В референсной системе, «общий код», состоит из трех частей:
  1) объектного каркаса абстрактных классов, задающего шаблонную структуру приложения и шаблонную структуру класса, реализующего стандартные операций над документом (поиск/ создание/редактирование/удаление/перемещение);
  2) набора вспомогательных методов и классов, предназначенных для построения пользовательского интерфейса в форме html документов;
  3) комплекса методов для обработки данных сложных html форм, кодирующих иерархическую структуру ОД, и поступающих в приложение в виде параметров CGI запроса, которые необходимо раскодировать и представить в виде иерархии объектов zFieldSet/zField. В эту же группу входят методы помогающие кодировать ОД, представленный zFieldSet/zField в виде html формы.

  Все эти части кода, а заодно и библиотека-аналог zDAO, были апостериори вычленены в результате реинженеринга монолитного приложения «SAM/Управление системой», после чего неоднократно и существенно дорабатывались. При проектировании «сверху вниз» можно, также как и в предыдущем случае, начать работу группой из трех человек, с одним лидером – *архитектором [прикладных] приложений*.

  1) *Архитектор приложений* должен будет заняться каркасом приложения и классами, задающими шаблонное поведение документа (частным случаем которого является тип ОД)
  2) первый программист, назовем его *дизайнер*, должен будет заняться внешним видом приложений и методами классов, для создания этого внешнего вида минимальными усилиями. Среди этих методов должны быть методы отображения разных *типов полей ОД*, в т.ч. словарных, когда значение поля подменяется описанием из словаря; и ссылок на другие ОД, когда вместо ZOID должен отображаться некий текстовый идентификатор ОД.
  3) второй программист, назовем его *трансформатор*, может заняться достаточно сложной задачей кодирования/декодирования (трансформации) данных ОД из zFieldSet/zField в html формы или XML документы и обратно. Использование XML представления позволит наладить обмен данными с другими системами, если в том возникнет нужда. Кроме того, с использованием XML представления можно реализовать универсальное приложение, позволяющее работать с любыми типами ОД, когда весь пользовательский интерфейс редактирования сводится к одному текстовому полю (редактору), содержащему XML представление ОД, и двух операций «Сохранить» и «Сохранить как новое». Такой интерфейс малопригоден для конечного пользователя, но он хорош для отладки и для программ-роботов. Кроме того, в современных браузерах возможно реализовать пользовательский интерфейс средствами JavaScript, обмениваясь с сервером документами в формате XML. Возможно что такое решение подойдет вашей системе больше, чем HTML 3.2 совместимый интерфейс.

  В любом случае, подготовив минимально необходимый «общий код», критически обсудив его в своем кругу, и подготовив каркасы для двух приложений, группа «А» должна разделиться: один (видимо *архитектор приложений*) остается документировать и систематизировать «общий код», а также отслеживать концептуальную целостность разрабатываемых приложений; второй (лучше - *трансформатор*) – должен заняться задачей «поддержка приложением одного типа ОД», на практике испытывая и доводя свои разработки; третий (*дизайнер*) – должен заняться приложением «SAM/Управление системой», где его ожидает встреча и совместная работа с членом команды

«K» (*безопасником*). Поскольку приложение «SAM» должно обслуживать несколько объекто-подобных сущностей – пользователь (с привилегиями и членствами в группах), группа доступа (с ACL), область данных (с SAMScopePolicy) и журнал аудита – эта задача также обозначена как распараллеливаемая. По завершению задачи «поддержка приложением одного типа ОД», появляется возможность строить пользовательский интерфейс всех остальных ОД по этому шаблону, при этом имеется три программиста (или даже четыре – включая *безопасника* из команды «K»), более-менее знакомых с этой технологией.

- D – «документация». Один технический писатель (хотя это уже зависит от количества и качества документов, которые необходимо написать). Программисты – плохие писатели, когда речь заходит о естественных языках, тем более, если этот естественный и родной язык не английский. Проблема в том, что языки программирования, изначально искусственные, с предельно конкретной и ограниченной лексикой, в процессе интенсивного ежедневного использования становятся на некоторое время основным языком мышления программиста. Сходную проблему можно наблюдать у людей изучающих иностранный язык или погружающихся в другую языковую культуру – сначала в разговоре начинают проскакивать иностранные слова, потом – чужеродные синтаксические конструкции, при этом человек затрудняется сформулировать свою мысль и на чужом и на родном языке. То же самое происходит и с программистом, после нескольких дней интенсивного кодирования. Вторая проблема – нежелание, в процессе документирования, повторно переживать сделанное, со всеми его недостатками. С системными архитекторами ситуация обстоит несколько лучше, но литературное качество их трудов бывает весьма графоманским. В результате, если хотите качественную документацию (особенно пользовательскую) – наймите технического писателя.
  На графике все задачи документирования слеплены в один большой столб, из которого не ясно что за чем следует, какое влияние документация оказывает на задачи кодирования и наоборот. Это подчеркивает параллельность процесса написания документации, процессу разработки. Не определено и то, что и чему предшествует – спецификация коду или код спецификации. Все это оставлено на усмотрение разработчиков, в какой форме они готовы принимать индивидуальные задания; как, кто и в каком объеме будет документировать результаты. Единственное правило, проведенное через все задачи документирования, если нет других предпосылок, то документация – это минимум 10% от кода описываемого объекта. В результате, для TIS из трех ОД, получился объем 7 тысяч строк, или 7000/50=140 страниц. Этот объем следует учитывать, как отдельную задачу по чтению и комментированию некоторыми участниками проекта (лучше – всеми).
- O – «библиотека zDAO». О возможности начать разработку с библиотеки zDAO сказано уже дважды. В рассматриваемом случае *архитектор системы* практически «прорывался» к этой точке через цепочку начальных задач по разработке ядра. Возможно, кому-то из команды «K» (прикладнику) надо было просто начать в точке «O», проигнорировав обозначенную зависимость от компонентов ядра, а *архитектору системы* – заняться разработкой задач «общий код ядра TIS» и «код ядра для одного ОД», с выходом на «кодогенератор для ядра».
- E – «эволюция», это самая интересная и перспективная точка входа в проект, поскольку именно в ней осуществляется добавление и модернизация типов ОД. Все что было написано ранее – это около 26 тыс строк общего код, обозначенного как «область T» (Рис. 14) и составляющего фундамент любой TIS, а с этого момента система начинает наполняться прикладной функциональностью. В этой точке *архитектор системы* должен подготовить спецификации типов ОД и осуществить порождение кода ядра и zDAO по ним. Полученный код для каждого типа ОД готов приблизительно на 80%, т.е. из 5 тысяч строк вручную пишется около 1 тысячи (0,2 в zDAO и 0,8.в ядре); еще порядка 1,6 – составляет «ручной »код прикладных приложений (действительный разброс составляет от 0,9 до 2,6 тысяч, что отчасти можно спрогнозировать по количеству таблиц). Так что с этой точки начинается довольно рутинная и предсказуемая деятельность. В среднем, добавление нового типа ОД в систему – это 2,6 т.строк «ручного кода», или 1-2 недели работы для двух программистов (одного в ядре и zDAO, второго – в прикладной области), после чего результат можно показать тестерам или пользователям, чтобы получить от них замечания. На этой стадии основное распараллеливание возможно за счет выделения каждому программисту задачи по написанию пользовательского интерфейса для одного или группы типов ОД.

Итак, проведенные выше спекуляции, позволили загрузить работой 6 программистов и 1-го технического писателя. Это достаточно большой коллектив, в котором много времени будет уходить на планирование и достижение взаимопонимания. Производительность у разных участников будет разной, что необходимо учитывать при распределении работ, чтобы «критические маршруты» прорабатывались наиболее надежными и производительными членами команды. При расчете календарного времени разработки по сетевому графику работ, могу предложить следующие эмпирические рекомендации:

- Зависимости задач друг от друга не являются абсолютными, и практически любую задачу в пределах «области Т» можно начинать не дожидаясь полного завершения всего, от чего она зависит. В каждой задаче следует определить минимальный набор функциональности, необходимый для начала работ от нее зависящих, и установить его достижение первоочередной целью.
- Время, необходимое на кодирование задачи, следует назначать в виде диапазона (*минимум;максимум*), где за *минимум* можно принять время при максимальной возможной производительности исполнителя (например 300 строк/день), а за *максимум* – при половинной (напр 150 строк/день). При этом можно задать условие, что по истечению *минимального времени решения задачи* должно быть выполнено все необходимое для начала работ над другими задачами, от нее зависящими.
- Для каждой задачи объемом более 0,3 [т.строк], к времени кодирования следует добавить 1 день до начала работы, для осмысления и планирования, а также 1 день после ее завершения, для оформления и описания сделанного.
- Для задач, имеющих зависимости от задач выполняемых другим членами команды, следует предусмотреть по одному добавочному дню на каждую такую зависимость. Этот день будет потрачен на ее изучение и понимание.
- В календаре разработки следует предусмотреть 5-6 *контрольных точек*(дней), когда никто не занимается кодированием. В эти дни каждый член команды готовит обзор сделанного им, после чего вся команда собирается вместе, чтобы заслушать эти обзоры и обменяться мнениями. Контрольные точки следует равномерно распределить по календарю и приурочить к завершению ключевых задач, таких как «простейшая TIS», «минимальный общий код ядра», «код ядра для одного типа ОД», «спецификация всех ОД», «Поддержка приложением одного типа ОД».

Примеры расчета календарного времени разработки по сетевому графику – приводить не буду, поскольку они громоздки и требуют большого числа спекулятивных допущений о качествах отдельных разработчиков и уровне взаимопонимания в команде. Каждый может проделать такой расчет самостоятельно, используя данные о своей команде и ее реальных характеристиках. Вместо этого приведу свою экспертную оценку «средней производительности одного программиста на стадии интенсивного кодирования», сделанную на основе нескольких таких расчетов, для вышеописанной команды из 6-ти программистов и ситуации близкой к идеальной – 100 строк в день. И теперь, уже на ее основе, привожу привожу оценку календарного времени разработки системы, подобной представленной на квази-ER диаграмме (Рис. 5), силами команды из 6-ти программистов:

- код «области Т» (25,6 т. строк): 25,6/0,1/6/22≈2 месяца
- «ручной» код для 2-х оставшихся типов ОД: (1+1,6)*2/0,1/6≈5 дней (т.е. 1 неделя)
- неучтенный вспомогательный код (10%): (25,6 + 2,6*2)*0.1/0.1/6≈5 дней
- Итого: **2.5 месяца** до стадии «работоспособная бета-версия» (впрочем, при таком коротком календарном сроке и большом количестве разработчиков – корректнее говорить о «работоспособной альфа-версии»)

Вообще, привлечение такого количества разработчиков одновременно, оправдано скорее для больших ИС, с несколькими десятками *типов ОД*, или в случае обучения новых членов команды используемому подходу, а также для отработки методов командной работы в «полевых» условиях. Для систем до 10-ти типов ОД – оптимальной будет команда из 3-х программистов и одного технического писателя, при времени разработки [с нуля] около полу-года. За это время, программный код «области Т» успевает лучше «выстояться» и «утрамбоваться», а результат получается более стабильным. Впрочем, все зависит от решаемой задачи и ее граничных условий.

На этом хочу завершить размышления о прогнозировании и распараллеливании процесса разработки, и сказать пару слов о его стабилизации, и той роли, которую в этом играет TIS/SQL подход.

При разработки документо-ориентированных систем на реляционной основе, после декомпозиции форм документов на реляционные таблицы, решение вопроса о создании пользовательского интерфейса для ввода и редактирования данных постоянно сталкивается с проблемой отнесения той или иной таблицы к тому или иному документу, поскольку это не следует очевидным образом из структуры БД. Проблема усугубляется в процессе развития системы, когда количество таблиц начинает расти, а структура БД все дальше и дальше отходит от «бумажной» основы (если она была). В этой ситуации возникают «кочующие» таблицы, постепенно мигрирующие из одного документа в другой, а в системе накапливается программный код, написанный разными программистами, в разное время, и имеющий разное представление об назначении и «положении» таких таблиц.

TIS/SQL подход диктует изначально сформулированную структуру документов, в которой базовые операции ведения и просмотра документов не вызывают разночтений. Изменение структуры документа является однозначным, и оказывает вполне прогнозируемое и контролируемое воздействие на другие типы документов, не затрагивающее их базовой функциональности. Перемещение таблицы из одного документа в другой – сопровождается изменением ее имени и структуры, что позволяет быстро выявить и исправить код, от нее зависящий. Распределение работ между программистами – упрощается, с самого начала имеются крупные и шаблонные задачи – проработка отдельных типов ОД. Таким образом, *типы ОД,* являются своеобразными зернами кристаллизации, вокруг которых формируется вся система. Опыт показывает, что структура БД выраженная в таблицах, в процессе разработки может измениться до неузнаваемости, в то время как набор *типов ОД*, и даже характер отношений между ними, – остаются практически неизменными.

Большую опасность для стабильности разработки (да и для стабильности системы) представляет СРД, если ее функционирование привязано к тематическим прикладным данным, и особенно если ее реализация «размазана» по прикладным приложениям. TIS/SQL подход абстрагирует СРД от прикладных данных, изолирует в компактном ядре, делая ее функционирование независимым от правильного или неправильного понимания прикладными программистами, которые могут вообще не задумываться об этой проблеме.

Проблемы, вызываемые сложными конкурирующими транзакциями, написанными разными программистами – локализованы в *ядре системы*, в виде хранимых процедур, а языки их написания в гораздо меньшей степени «затеняют» проблемный алгоритм, чем универсальные языки программирования. Сложность работы с БД через хранимые процедуры локализована в библиотеке zDAO, которая к тому же предоставляет (и диктует) прикладным программистам способ мышления в терминах объектов предметной области.

Изменения структуры БД, такие как добавление/удаление полей и таблиц, в значительной степени обслуживаются *кодогенератором*, избавляя систему от ошибок привносимых программистами. То же самое касается и изменений базовых алгоритмов манипулирования данными таблиц и правил функционирования СРД, которое должно быть одномоментно встроено во все хранимые процедуры.

5.2. Правила проектирования и внесения изменений

Суммируя и систематизируя все сказанное в этой работе о проектировании систем в рамках TIS/SQL подхода:

1. Типовая TIS состоит из трех подсистем: СРД, словарь и «прикладная область». Проектирование каждой из них подчиняется своим правилам.
   1.1. СРД и словарь проектируются в рамках стандартного реляционного подхода, при этом для СРД доминирующей идеей является использования числовых суррогатных первичных ключей (id), а для словаря – составных символьных ключей-кодов {`ZCode.dic, ZCode.code`} (Таб. 17). Более фундаментальное различие состоит в том, что данные СРД считаются динамическими и изменяются через пользовательское приложение, а данные словаря – статическими, относящимися скорее к структуре БД, и изменяемыми с привлечением администратора БД.

1.1.1. При проектировании СРД главным рабочим документом является ER-диаграмма, а для каждой таблицы необходимо составить формальное описание (например по форме Таб. 18). Помимо этого, для СРД необходим документ, описывающий принципы работы, включающий рекомендации по ее встраиванию в остальные части ядра, а также последующему использованию и настройке в разрабатываемой ИС. Спецификацию хранимых процедур *ядра системы*, обслуживающих СРД, можно оформить в виде приложения к такому документу.

1.1.2. При проектировании словаря, также как и для СРД составляется ER-диаграмма и готовятся формальные описания таблиц, но документ, описывающий принципы использования словаря, является лишь кратким, конспективным изложением соответствующих фрагментов документации на *тип ОД*, для которого те или иные таблицы введены в словарь.

1.2. Проектирование в прикладной области, после реализации фундаментального кода «области T», сводится к проектированию *типов ОД* и связей между ними.

1.2.1. Главным рабочим документом здесь является квази-ER диаграмма (Рис. 5), для каждой таблицы составляется формальное описание (по форме Таб. 18), а для каждого типа ОД – документ, описывающий его семантику, способ применения, отношения записей внутри ОД, связи с другими ОД, критерии качества и используемые таблицы словаря. Это описание является исходным для разработки процедур *контроля качества*.

1.2.2. Проектирование каждого типа ОД можно рассматривать как отдельную задачу, но в этом случае потребуется их последующее согласование в рамках квази-ER диаграммы.

1.2.3. В больших ИС, состоящих из несколько подсистем, можно проектировать их раздельно, составляя квази-ER диаграмму на каждую подсистему в отдельности, схематично показывая связи с «чужими» типами ОД. При таком проектировании целесообразно ввести понятие «код подсистемы», который можно использовать в качестве префикса к «имени типа ОД» и «коду типа ОД», что сделает независимыми пространства имен каждой из подсистем. Возможно что вместо префикса к именам таблиц (в виде кода подсистемы), их следует помещать в отдельную схему БД (SQL SCHEMA), соответствующую этой подсистеме. Имя схемы может совпадать с именем подсистемы или ее кодом, а в СРД, там где это необходимо (например в ACL), будет использоваться «код подсистемы», в виде отдельного поля или в качестве префикса к другому коду.

2. Внесение изменений в TIS должно следовать по тому же пути, что и проектирование, т.е. в начале изменению подвергаются документы: ER и квази-ER диаграммы, описания таблиц и типов ОД. Изменения в программный код системы вносятся по результатам согласованного пакета корректив к рабочим документам, с четким осознанием их причин и последствий.

2.1. Прежде всего необходимо подготовить документ – «проект изменений» – кратко (2-3 абзаца) описывающий потребности, удовлетворению которых они должны послужить, более подробно (3-4 абзаца) – что это за изменения, и максимально подробно – какие коррективы будут внесены в каждый документ. Последнее должно быть сделано так, чтобы изменения этих документов сводилось к нетворческому исполнению инструкций, а еще лучше – копированию в них измененных фрагментов. Ключевой частью «проекта изменений», по которой ведется его обсуждение и исполнение, являются ER и квази-ER диаграммы, на которых обозначены вносимые изменения (Рис. 15, сравни с Рис. 5).

2.2. Далее, этот «проект изменений» подвергается критическому обсуждению и согласованию в рабочей группе, состоящей из системного архитектора и разработчиков, которых затрагивают предложенные изменения, с возможным привлечением представителя заказчика. Здесь определяется объем работ, назначаются ответственные, сроки исполнения и контрольные точки.

2.3. Внесение изменений в программный код состоит из следующих стадий:

2.3.1. Генерация автоматического кода

2.3.2. Написание скрипта трансформации структуры БД в новое состояние, без потери данных. В TIS эта задача осложняется хранением истории изменений, что требует принятия некоторого компромиссного решения о том, как должны выглядеть устаревшие версии ОД, с учетом новой структуры таблиц. Если такая трансформация ведет к утрате части исторических

данных, это должно быть отражено в инструкции администратору системы, который должен будет выполнить специальную резервную копию БД, хранимую весь срок жизни ИС.

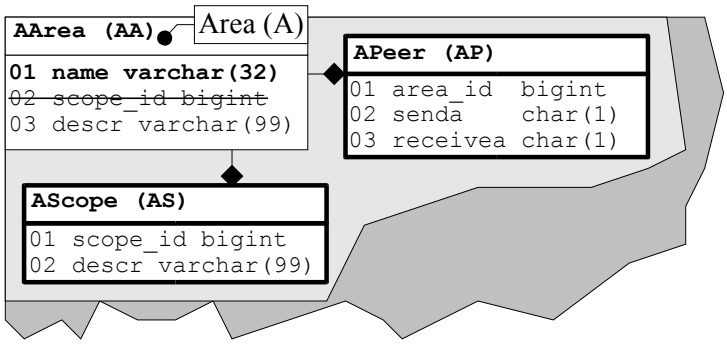
2.3.3. Внесение изменений в «рукописные» процедуры (или написание этих процедур).

2.3.4. Внесение изменений в zDAO и прикладные приложения.

2.3.5. Комплексное тестирование.

2.4. Внесение изменений должно завершаться порождением новой версии ПО ИС и написанием документа: «Инструкция администратору системы по переходу ИС с версии X на X+1»

NB Потребность в изменение структуры данных, для многих ИС, находящихся в эксплуатации, – является регулярной. Это требует отработанной и рутинной процедуры их совершения, отладки и инсталляции в работающую систему в виде обновлений. Если говорить о разработке ИС, то минимум половина времени, начиная с достижения «высшей точки» и до конца проекта, тратится на такие изменения, что предоставляет хорошую возможность отработать эти процедуры со всех сторон (технической, экономической, организационной).



Задача_1: Ввести множественность scope для каждой Area
Решение: таблица AS вместо AA.scope_id. перекрытие Area по scope - запретить.
Задача_2: управление разрешенными направлениями приема/передачи ОД, в зависимости от Area
Решение: введение таблицы APeer, где area_id – партнер, AP.senda – разрешена отправка в (Y/N), AP.receivea – разрешен прием из.

Рис. 15 "Проект изменений" на квази-ER диаграмме (пример)

5.3. Правила составления и написания имен и идентификаторов

Во всех проектах складываются свои «птичьи» языки и «живые» словари терминов «широко известных в узком кругу». В рамках TIS/SQL, также накопилось достаточно количество постоянно используемых лексем, имеющих устойчивое написание и особую семантику: ZOID,ZRID, ZNAME, ZN, ZO, SAM, SAMUser и т.п. Одновременно, в процессе эволюции этого подход к разработке ИС, сложились некоторые морфологические законы – правила по которым выбираются новые и составляются производные идентификаторы для объектов БД (таблиц, полей, процедур), имен переменных, синонимов, функций, классов и методов классов.

По все этой работе рассредоточено множество лексем, претендующих на устойчивое положение в «словаре» TIS/SQL, и достаточное количество примеров словообразования, на которые можно ориентироваться. При этом необходимо осознавать, что словари и правила не есть нечто незыблемое и застывшее, это скорее языковая основа для формирования новых внутри-проектных языков, которые будут интенсивно пополняться новыми лексемами и морфемами, соответствующими прикладной области, равно как и новыми способами и шаблонами их использования.

Важно, чтобы все участники проекта имели «совместимое» понимание этих правил, что облегчает их взаимодействие и перекрестное чтение кода. Наличие общего происхождения у «языков» разных проектов, в дальнейшем, должно благоприятно сказаться на взаимопонимании членов разных команд, в процессе интеграции систем разработанных независимо, или при объединении их усилий для работы над новым проектом.

В свете вышесказанного, привожу свод базовых морфологических правила «по манифесту»:

1. Имена самостоятельных сущностей (*корневые имена*) – имена для типов ОД («Пространство», «Документ», «Контейнер»), подсистем («СРД») и любых других самостоятельных понятий («Транзит», «Словарь») – обычно заимствуются из английского языка, поскольку им более-менее владеют все программисты, вне зависимости от национальности. Обычно эти имена – существительные в единственном числе, или иные части речи и словосочетания, используемые в роли существительного: `Area, Document, Container, SAM, Transit, Dictionary`. Корневые имена всегда пишутся с заглавной буквы, если же они являются аббревиатурами – то все буквы пишутся заглавными. Корневые имена могут быть составными словами, в этом случае слова пишутся слитно, но каждое – с заглавной буквы (например `ObsoleteRule`).
   1.1. Можно, из идеологических соображений, воспользоваться в качестве основного источника заимствований любым другим языком, например латынью, греческим, эсперанто или вообще – родным языком разработчиков. Главное, чтобы полученные идентификаторы транскрибировались и записывались стандартным латинским алфавитом, без диакритических знаков (т.е. доступным в коде ASCII).
   1.2. Все *корневые имена* уникальны для всей ИС. Длина *корневого имени* не регламентируется.
   1.3. Необходимо осознавать, что сразу после заимствования смысл исходного слова подвергся трансформации и исказился – теперь оно обозначает конкретное явление разрабатываемой ИС, и пора задуматься о составлении глоссария специализированных терминов.
   1.4. Для простых вещей следует выбирать простые, часто используемые слова. Сложность явления или понятия можно подчеркнуть редким словом или заимствованием из другого языка.
   1.5. Корневые имена в чистом виде используются на стадии проектирования, в документации, иногда в качестве имен директорий и имен классов. Во всех остальных случаях используются производные от них идентификаторы, образованные с помощью аффиксов (префиксов, постфиксов, расширений и т.п.)
2. *Коды*. Для каждого имени, вводится его синоним – код фиксированной или ограниченной длины. Код уникален на том же пространстве, что и имя, для которого он введен. Код – это всегда заглавные буквы и/или цифры (`ZO, ZN, XUG`).
   2.1. Коды являются аналогами местоимений. Область их применения – синонимы имен таблиц в сложных SQL запросах (`select AP.ZOID, AA.area_id, AA.name from AArea AA, APeer AP where AP.area_id = AA.ZOID`); имена локальных переменных; префиксы к именам и кодам сущностей, подчиненных обозначаемой кодом; указание на тип объекта (ОД, таблицы, поля) для СРД (ACL) и аналогичных случаев; аффиксы в именах процедур и функций, указывающие на их принадлежность и специализацию (`commit_A, list_refs_AP`).
   2.2. Полный код, составленный сложением всех кодов сущностей в порядке их подчиненности, – уникален для всей ИС. Например: для типа ОД «`Area`» – код «`A`»; для таблицы `APeer` - «`AP`»; для поля `APeer.area_id` - «`AP01`». Фиксированность длин составных частей кода – облегчает его деление на составные части.
3. Имена подчиненных сущностей (таблиц, полей):
   3.1. Имена таблиц (типов записей) – образуются по тем же законам, что и корневые имена, но с приписыванием в качестве префикса *кода типа ОД*, что позволяет использовать одни и те же корневые слова для разных таблиц, относящихся к разным типам ОД или подсистемам. Например: `AArea, APeer;` или вообще: `AHeader, ARow, BHeader, BRow`.
   3.2. Имена таблиц всегда пишутся с нескольких заглавных букв – все знаки, вплоть до первой буквы корневого слова включительно – заглавные. Если корневое слово составное – каждое слово пишется с заглавной буквы. Например: `SAMUserCapability, SAMScopePolicy`.
   3.3. Имена полей образуются заимствованием слов из основного языка, однако пишутся строчными буквами. Составные слова пишутся слитно, но иногда в качестве разделителя используется символ «_». Достаточно распространено сокращение заимствованных слов («`descr`» вместо «`description`», «`obj`» вместо «`object`»).
   3.4. Для имен полей применяется аффиксация. Вот наиболее устоявшиеся морфемы: префикс «`is_`» - указывает на булевый тип поля-характеристики (`is_local`); постфикс «`_id`» - на поле-ссылку (`obj_id`); аффиксы «`type`» и «`code`» - на поле, требующее словарного значения; постфикс «`a`», образованный от «`allowed`» - на то, что значение этого поля контролирует

некоторое разрешение или запрет (`writea`); префикс «`do`» - на флаг, установка которого задействует некоторую функциональность системы или объекта (`doaudit`).

4. SQL не чувствителен к регистру имен таблиц и полей, однако регистр важен для большинства прикладных языков программирования, при этом крайне желательно единство написания этих идентификаторов во всех частях системы. Эталоном написания, с точки зрения регистра, являются формальные описания таблиц, отраженные на основных рабочих документах – ER и квази-ER диаграммах. Поля стандартных заголовков (`ZOID, ZRID, ZPID` и т.д.) всегда и везде должны записываться заглавными буквами.
5. Образование кодов.
   5.1. В качестве кода типа ОД используется первая буква имени типа, если она еще не занята. Иначе – выбирается фонетически наиболее сильный звук (например для Container это "T"), иначе – любой свободный символ. Если в системе планируется большое количество типов ОД – следует перейти на двух или трех буквенные *коды типов ОД* и/или на уникальность имени и *кода типа* в пределах подсистемы, имеющей собственный код.
   5.2. При образовании кодов подчиненных сущностей (например таблиц), полный код складывается из кода вышестоящей сущности (например типа ОД) и добавочного кода подчиненной сущности, уникального среди ее «соседей» по подчиненности. Например: ОД Area (код A), таблица AScope (добавочный код S, полный код таблицы – AS).
   5.3. Добавочные коды таблиц образуются от их корневых слов, по аналогии с кодами типов ОД.
   5.4. Для полей, в качестве добавочных кодов, используются код из двух цифр, похожий на порядковый номер поля в таблице, указываемый в описаниях таблиц и на квази-ER диаграмме. Так код поля «AArea.descr» - «AA03».
   5.5. Для кодов подсистем рекомендуется трех-символьная форма записи. (Например: SAM - «Система разграничения доступа»; BIL - «Бухгалтерия»; EMP - «Сотрудники»; NET - «Сетевая инфраструктура»; STO - «Склады» и т.п.)
   5.6. Следующие специальные коды типов ОД зарезервированы: Z – системные данные; X – таблицы СРД; Y – символ-расширитель (префикс), позволяющий ввести многосимвольные коды типов ОД, с сохранением существующих односимвольных; S – однотабличные, необъектные пользовательские сущности (см Таб. 5).
6. Имена процедур, функций и методов. Эти идентификаторы являются, обычно, императивными высказываниями, состоящими из глагола, обозначающего действие, и нескольких слов его уточняющих. Язык заимствования тот же, что и для всех остальных идентификаторов.
   6.1. Форма записи словосочетания может быть двоякой, например: «move_object» или «MoveObject». Первый вариант рекомендуется для имен хранимых процедур, а также для функций прикладного ПО, находящихся на низком уровне абстракции («близких к земле»). Второй вариант подходит для имен методов классов, находящихся на более высоком уровне абстракции, на котором метод MoveObject может инкапсулировать различные реализации для различных случаев: вызов move_object() в одном; вызовы send_object() и receive_object() в другом; обращение к удаленному серверу RPC/CORBA/EJB/DCOM в третьем.
   6.2. В именах функций и методов, обрабатывающих или возвращающих множества объектов, имена-идентификаторы должны использоваться в единственном числе. Например: `SearchObject, ListObject` вместо (Search|List)Objects. В крайнем случае, можно подчеркнуть множественность служебным словом «`Set`»: `DeleteObjectSet(SomeSet)` в значении «удалить переданный набор объектов».
7. Множественное число. Рекомендуется образовывать множественное число только аналитически, присоединяя к идентификатору слово «List» или «Set», например: `ObjectList`, употребляется в значении «list of objects» или «objects». Обычных средств образования множественного числа в основном языке заимствования следует избегать, поскольку они могут быть иррегулярны и затруднить автоматическую кодогенерацию в будущем.

Производные идентификаторы (имена классов и полей классов) в библиотеке zDAO и прикладных приложениях образуются от имен соответствующих типов ОД, таблиц и полей. При необходимости, имена классов снабжаются «венгерскими» префиксами, подобно тому как это показано на Рис. 10.

5.4. Место и роль автоматической кодогенерации

Через весь манифест, от начала и до конца, красной нитью проходит идея автоматической генерации кода по формальным определениям *типов объектов данных* и таблиц, входящих в их структуру. В каждом разделе, при каждой возможности, указывались те части системы, для которых создание программного кода следует поручить кодогенератору. Настало время расставить все точки над i и трезво взглянуть на эту задачу, избавившись от определенных иллюзий «первого взгляда».

Прежде всего проанализируем некоторые количественные данные, которые можно извлечь из Таб. 22, вспомнив о том, что кодогенерации хорошо поддается только код *ядра системы* и библиотеки zDAO, относящийся к «прикладной области», и о том что готовность этого кода порядка 80%. Простейший расчет показывает, что объем кода, находящегося под управлением кодогенератора, для референсной системы из пяти типов ОД составляет всего 26%:

```
(24.4-8)*0.8/50*100% ≈26%
```

если же учесть, что из пяти типов ОД код одного должен быть написан вручную, да и сам кодогенератор (1.6 т строк) требует разработки, то его КПД падает до 18%.

```
((24.4-8)/5*4*0.8-1.6)/50*100% ≈18%
```

> **NB** термин *КПД кодогенератора* неудачный, но звучный и интуитивно понятный, поэтому вместо поиска более «правильного», просто определим его как: «отношение объема кода, построенного кодогенератором, к общему объему кода системы».

Экстраполируя полученные данные, становится очевидным, что даже в очень большой системе, при стремлении количества типов ОД к бесконечности, КПД кодогенератора имеет предел: приблизительно 54% по данным таблицы Таб. 22, и 61% с учетом поправок на различия между референсной системой и TIS (для последней КПД достигает 50% при количестве типов ОД 25, что соответствует довольно сложному проекту). В реальных ИС, где потребности могут далеко выходить за рамки стандартных операций над ОД, с которыми хорошо справляется кодогенератор, его КПД может оказаться значительно меньше. Таким образом – кодогенератор не есть замена программистам.

Действительно, говоря о кодогенераторе, следует вспомнить об *описаниях типов ОД*, по которым он работает, и условном языке, на котором эти описания составлены. Фактически, речь идет программировании на специализированном языке более высокого уровня и трансляторе с него, на реальные языки программирования, используемые в проекте. Впрочем, нормы этого условного языка (назовем его TIS/DDL) пока еще очень подвижны, равно как и код «транслятора». Подвижность этих норм является неотъемлемой частью понятия «кодогенератор» в TIS/SQL подходе на данной стадии.

Главная причина, по которой TIS/SQL требует обязательной машинной генерации любого шаблонного кода по формальным определениям типов ОД и таблиц – необходимость обеспечить простоту и безошибочность модификации всех зависимых компонентов (view, процедур и классов) при изменении описаний типов ОД, их таблиц и полей, а также при изменении шаблонов по которым эти компоненты должны быть написаны. Все прочее – приятные, но побочные качества.

Применение кодогенерации для пользовательского интерфейса – возможно, но польза от этого – сомнительна, поскольку здесь часто требуется творческая изощренность, недоступная автомату.

Результатом работы кодогенератора всегда являются файлы, инсталлируемые в заданные ветви дерева исходных текстов проекта. В общем случае, эти файлы никогда не редактируются программистом, любые изменения в них вносятся только инсталляцией нового варианта, порожденного кодогенератором. На вопрос: «Что должно помещаться в CVS репозитарий – формальные описания или файлы построенные по ним кодогенератором?», мой ответ – и то, и другое. При тестировании или сертификации – предметом исследования будут тексты на реальных языках.

> **NB** Кодогенератор очень далек от качества полноценной программы, и должен оставаться таковым - довольно грубым, легко-модифицируемым скриптом, написанном на подходящем для подобных задачи языке (напр. awk, в крайнем случае – perl). Не следует добиваться совершенства, пытаясь создать из него некий самостоятельный продукт, если это не является осознанной и экономически обоснованной целью. Его главные качества для успеха конкретного проекта – дешевизна разработки и легкость модификации (оптимум результата при минимуме затрат).

6. Заключение

Итак, настало время подведения итогов всему сказанному в этой работе, а лучшие итоги — это перспективы новых работ, следующие из проделанной. Предметом этой работы стала та фундаментальная составляющая ИС, которая должна закладываться на самой ранней стадии разработки, и для тщательной проработки которой, в большинстве случаев, не хватает ни времени ни ресурсов. «Манифест» декларирует систему базовых понятий, принципиальную конструкцию системы, подробно описывает необходимые ей структуры данных и, по каждой из затронутых тем, приводит перечисление вопросов и проблем ей сопутствующих, делая упор не столько на их однозначном решении, сколько на указании тех путей мысли, где эти ответы и решения могут быть найдены, последовательно, в порядке усложнения, начиная с простейших. Завершающие главы работы дают представление о возможных конструкциях реальных систем и затратах на их реализацию. В целом, несмотря на конкретность и практическую ориентированность изложения, в нем сильна философская составляющая, диктующая не только и столько конкретные решения, сколько образ мысли, из которого они следуют и в рамках которого должны эволюционировать далее, в процессе адаптации к конкретным задачам.

Для дальнейшего развития темы этой работы следует реализовать референсную информационную систему, воплощающую основные идеи и структуры данных «манифеста». В качестве проекта для такой ИС, целесообразно воспользоваться квази-ER диаграммой Рис. 5, для которой уже приведена оценка трудозатрат, и которая демонстрирует основные идеи TIS/SQL подхода, не обременяя его частными проблемами реальных задач. Исходные тексты такой ИС станут основой и отправной точкой для быстрой разработки прототипа системы под любую реальную задачу. Исходные тексты референсной ИС целесообразно открыть для свободного и безвозмездного использование любым потенциальным потребителем — разработчиком ИС. В качестве дальнейшего развития этой концепции продвижения идей TIS/SQL, возможно разработать и опубликовать более сложную ИС, имеющую практическую ценность для некоторого слоя потенциальных потребителей, обладающих потребностями в ней и возможностями по ее развитию. Скорее всего это должна быть система управления предприятием (учет кадров, материальных ресурсов, договоров и отслеживание выполняемых работ) или система электронной коммерции (например web-магазин для торговли через Internet). Именно такие системы, в современных условиях, обладают перспективами получить большое число инсталляций, каждая из которых может стать «точкой роста», вокруг которой сформируется коллектив разработчиков, ангажированных на использование и критическое осмысление TIS/SQL подхода.

Отдельной референсной реализации требует концепция распределенных и репликативных ИС, которая должна быть получена из «одиночной» системы, разработанной по квази-ER диаграмме Рис. 5, и снабжена концептуальным документом, описывающим процесс трансформации системы с единой БД в распределенно-репликативную. Реализация полноценной функциональности, обслуживающей репликацию, в первом же варианте референсной ИС – нецелесообразна, поскольку он должен быть максимально прост для понимания, а кроме того – проблема трансформации заслуживает отдельного описания.

На основе рабочей документации референсных ИС, ориентируясь на сложившейся круг потенциальных читателей, можно будет подготовить и опубликовать ряд работ, более детально раскрывающих различные аспекты разработки, эксплуатации и развития ИС в рамках TIS/SQL подхода.

Еще одной целевой аудиторией TIS/SQL подхода являются разработчики ИС для государственных органов и предприятий, работающих с информацией, составляющей гос-тайну. Скорее всего, в современных условиях, эти разработчики будут приходить из сферы разработки и сопровождения коммерческих ИС и сферы Internet систем.. В любом случае, наличие успешных применений и сформировавшегося сообщества разработчиков в этих средах — должно благоприятно сказаться и на продвижении TIS/SQL в других отраслях, и на понимании разработчиками проблем и способов защиты информации в сложных ИС.

В более отдаленной перспективе, при успешном осуществлении вышеперечисленного, возможно провести оптимизацию некоторых СУБД для использование в качестве основы TIS/SQL, или провести разработку новой, в которую будет изначально встроена требуемая система понятий и оптимальные алгоритмы ее обеспечивающие.

С уважением. Евстропов А.В. (aka Alex V Eustrop). Москва. май 2008

Приложение А Происхождение TIS/SQL, системы-предшественники

Ничто не создается на пустом месте. В любой конструкции или идеологии более или менее явно просматриваются идеи и целые фрагменты ее предшественников, а проблемы ею решаемые, решены только потому, что возникли ранее, в процессе создания или эксплуатации этих предшественников. Здесь я хочу ретроспективно рассказать, обо всех системах, потомком и наследником которых является TIS/SQL, с указанием заимствования идей и проблем.

Мое участие в этих разработках началось в 1993 году, с проекта NUMACS, поэтому о более ранних разработках мои сведения, полученные с чужих слов, могут быть неполны и неточны. Впрочем о проектах, в которых я участвовал, они будут не только неполные и неточны, но еще и субъективны, и предвзяты.

Прежде чем приступить к описаниям этих проектов, хочу акцентировать внимание, большая часть систем предшественников, на протяжении более чем 30-ти лет, разрабатывалась (и продолжает разрабатываться) под руководством и при непосредственном участии Румянцева Александра Николаевича, начальника Отдела Информационных Систем (ОИС), НТК-Электроника, РНЦ «Курчатовский Институт». Собственно говоря, на почве сформулированных и реализованных им идей, а также возникших при этом проблем и разногласий, зародился и сформировался TIS/SQL подход, как некоторая идеология и методология, претендующая на самостоятельную ценность.

197x система обработки отчетов для МАГАТЕ (http://www.iaea.org/)

Система ввода хранения и обработки отчетов с ядерных установок, поступающих в МАГАТЕ, в рамках соответствующих международных соглашений. Система разработана на базе ЭВМ серии IBM360/390, производства IBM®, СУБД ADABAS™, производства Sofware AG, а основным языком программирования являлся PL/1. Это была пакетная система, отчеты поступали на бумажных носителях и вводились в систему вручную, что автоматически поднимало проблему создания многоступенчатого контроля качества, развитой системы коррекции введенной информации и устойчивости к любым программно-аппаратным сбоям. В этом проекте впервые остро встал вопрос хранения истории всех изменений (коррекций), с целью разбирательства любых спорных ситуаций и воспроизводимости отчетных документов на любую дату в прошлом. Значительную часть этой ИС спроектировал и написал Румянцев А.Н. Система находится в эксплуатации до сих пор.

Ключевая характеристика СУБД ADABAS, используемая всеми информационными системами на его основе – сложная структура данных, позволяющая поместить весь документ в одну запись и обеспечить, очень высокую скорость обработки (как в ненормализованных реляционных БД, при последовательной обработке больших массивов записей). В некотором смысле, идеология ADABAS похожа на идеологию объектные СУБД, развивающуюся в наше время.

198x – 199x МИС (КИАЕ, http://www.kiae.su)

«Мета-ядро Информационных Систем» (МИС) явилось реимплементацией основных продуктивных идей системы для МАГАТЕ, выполненное самим Румянцевым А.Н. и его сотрудниками, после возвращения в «Курчатовский Институт Атомной Энергии» (КИАЕ). Программно-аппаратные средства: PL/1, ЭВМ серии ЕС (аналоги IBM360/390), СУБД ADABAS. МИС – это прежде всего идеология и инфраструктура, на базе которой в кратчайшие сроки можно было создавать и запускать в эксплуатацию системы для ввода, хранения и обработки информации любого назначения. Ключевыми частями МИС являлись способы хранения и коррекции информации, позволяющие хранить полную историю изменения данных, с воспроизводимостью отчетных документов на заданную дату и некоторый унифицированный конвейер поступления информации в систему, состоящий из следующих стадий:

- система терминального набора документа (пакета или «лота», нового или «коррекции» к сущ.)
- Постановка «лота» в очередь на обработку «загрузчиком»

- Обработка «лота» «загрузчиком», и помещение его в БД, в пред-обработанном виде
- Обработка «лота» специализированным процессом «контроля качества»
- Загрузка и актуализация изменений в БД

На любой стадии, обработчик может сформировать обоснованный отказ, в результате которого документ возвращается на доработку (обычно – на самую первую стадию).

МИС, в качестве значений полей, широко использовал символьные, мнемонические коды, которые могли без расшифровки присутствовать как во входных так и в результирующих документах ИС. Принципы символьного, мнемонического кодирования, возведенные в догму TIS/SQL, а также некоторые служебные мнемо-коды (например "N" и "C" для поля ZSTA) уходят корнями в МИС.

В 1986 г, МИС использовался для экстренного создания системы сбора информации о радиационном заражении местности после аварии на Чернобыльской АС, и вероятностного моделирования его распространения.

Последняя из «МИС» систем была выведена из эксплуатации в конце 1990-х годов, вместе с ликвидацией парка ЕС-ЭВМ.

1992 – 1994 NUMACS (РНЦ «КИ», http://www.kiae.ru)

«Nuclear Material Accounting Control and Safeguards» (NUMACS) – это фундаментальный, двух-томный труд, посвященный принципам построения системы учета ядерных материалов (ЯМ) и контроля за их сохранностью, основанной на принципах измеряемого материального баланса, с учетом физической специфики учитываемых веществ. NUMACS являлся концептуальным проектом по созданию такой системы для Российской Федерации. Основные авторы этой работы – сотрудники РНЦ «КИ», когда-то работавшие в МАГАТЕ, и обобщившие своей многолетний опыт в этой сфере: Румянцев А.Н., Сухоручкин В.К. и Шмелев В.М.

На основе NUMACS, была разработана прототипная информационная система, демонстрирующая его основные идеи, с применением которой была проведена первая физическая инвентаризация рассекреченных ЯМ на двух ядерных установках Курчатовского Института, «НАРЦИСС» и «АСТРА». ИС получила условное название, по аббревиатуре омонимичное проекту NUMACS – «Nuclear Material Accounting and Control System».

Разработка произведена ОИС (РНЦ «КИ», НТК-Электроника), под руководством и при непосредственном участии Румянцева А.Н., начальником Группы Администратора Баз Данных (ГАБД) Остроумовым Ю.А. и мною, сотрудником этой группы Евстроповым А.В. Моей частью была разработка пользовательского интерфейса для ввода, просмотра и редактирования данных, в соответствии с принципами организации данных МИС. Однако эта система уже не была частью МИС, она разрабатывалась как интерактивная система с архитектурой клиент-сервер, на базе СУБД ADABAS, функционирующем под управлением IBM OS/2™, и приложений в среде NATURAL™, функционирующей на клиентских рабочих станциях под управлением ОС Microsoft Windows™ for Workgroup 3.11. В данной ИС не уделялось особого внимания защите информации и разграничению доступа к ней.

Далее, все работы по теме «Учет и Контроль ЯМ» проводились в рамках межлабораторного сотрудничества между РНЦ «КИ» и Американскими Национальными Лабораториями во главе с Los Alamos National Laboratory (http://www.lanl.gov). Результатом этого явилась полная смена базовых технических средств, на которых осуществлялась разработка «Системы Учета и Контроля ЯМ» для РНЦ «КИ». Состав этих средств – MS Windows NT™, MS SQL Server™, MS Visual Basic™. Новая система получила название KI-MACS (Kurchatov Institute Material Accounting and Control System).

1995 – 2001 KI-MACS (РНЦ «КИ», http://www.kiae.ru)

ИС KI-MACS стала дальнейшим развитием идей проекта NUMACS не только технически, но и концептуально. Она также явилась попыткой адаптации концепции МИС к абсолютно новым

программно-аппаратным средствам, потребностям пользователей и условиям эксплуатации. Технически KI-MACS представляет собой систему клиент-сервер, где на стороне сервера функционирует БД под управлением MS SQL Server, а на стороне клиента – оконные приложения, написанные на Visual Basic, и взаимодействующие с сервером через ODBC (инкапсулированное в объектную библиотеку RDO). Фундаментальными отличиями от любых подобных систем стали унаследованная от МИС идеология и методология ведения полной истории всех изменений, и, введенная для целей разграничения доступа, полная изоляция таблиц БД от пользовательских приложений комплексом хранимых процедур – «ядром системы», реализованным без использования view, что ставит KI-MACS значительно ближе к «трех-звенной» архитектуре, получившей распространение позже, чем к классическому пониманию «клиент-сервер» для реляционных СУБД. Фактически, система утратила большую часть характерных свойств реляционных БД и приложений, их использующих.

В KI-MACS была реализована весьма изощренная система разграничения доступа, привязанная к разнообразным прикладным данным, зачастую косвенным образом. Среди прикладных данных, используемых СРД, присутствовали поля, прямо и косвенно определяющие принадлежность документов (объектов) конкретной ядерной установке, и поля, указывающие их уровни секретности.

В результате система была сертифицирована Гостехкомиссией России (ГТК РФ) по классу «1В» (один «Вэ»), что позволяет использовать ее для обработки информации составляющей государственную тайну. (Класс «1В», по требованиям ГТК РФ, позволяет системе хранить и обрабатывать информацию различных грифов секретности одновременно, и концептуально близок омографичному классу «1В» (one «Bi») по т.н. «Оранжевой книге» Мин. Обороны США).

Система разработана в ОИС (РНЦ «КИ», НТК-Электроника), под руководством и при непосредственном участии Румянцева А.Н. В разработку был вовлечен весь коллектив отдела, а также временные сотрудники и сотрудники других подразделений. Общее число сотрудников, в разное время задействованных в разработке и тестировании 18, из них непосредственно в написании кода – 8. Большая часть работы, особенно в первые годы, была выполнена Румянцевым А.Н. и мною, при этом разделение областей ответственности было произведено по естественной границе между «ядром системы», которым занимался только Александр Николаевич, и «прочим, клиентским ПО», которым занимались все остальные, в т.ч. и я.

В процессе разработки дизайн системы претерпел многократные, иногда кардинальные изменения, одно из которых представляло собой переход от концепции одиночной БД к концепции «распределенных» БД, обменивающихся между собой информацией о проведенных транзакциях. (Последнее явилось одной из исходных проблем, под решение которой сформулирована TIS/SQL концепция «области данных».)

В значительной степени кардинальные повороты дизайна были обусловлены разработкой в условиях «первопрохождения», когда реальные проблемы и пути их решения неизвестны заранее. Переосмысление этих проблем, способов их решения, а также вторичных проблем следующих из того или иного решения легло в основу моих последующих работ, и в результате сформировало TIS/SQL подход.

Вторая «действующая сила постоянных перемен» – желание создать систему, потенциально удовлетворяющую потребности любой ядерной установки и любой организации на территории РФ, в условиях неосознанности этих потребностей на уровне потребителей, и недостаточной проработанности на уровне законодательства.

Мое участие в проекте KI-MACS закончилось в 2001 году, написанием 106-ти страничного монографического документа «KI-MACS FrontEnd. Руководство программиста», описывающего архитектуру прикладных приложений, разработанных мною (порядка 70 тыс. строк), и на основе которых создавались все остальные пользовательские приложения KI-MACS. Многие идеи, описанные в этом документе получили свое развитие в следующем проекте, YELL, и в конечном результате стали частью TIS/SQL. В частности термин «объект данных», для совокупности записей составляющих единый документ, был впервые введен именно в прикладных приложениях KI-MACS, на уровне объектной библиотеки KDO (KI-MACS Data Objects), являющейся предтечей zDAO, рекомендуемой TIS/SQL подходом.

2001 – 2004 YELL, разработка для РосНИИРОС (http://ripn.net)

Мое появление в команде разработчиков РосНИИРОС, осенью 2001-го приступавших к работе над новым проектом, было незапланированным и почти случайным – тем не менее, именно мои идеи и принципы построения документарных систем, в конце-концов, стали его основой. Новая ИС должны была объединить и заменить две разрозненные (и очень разные БД), эксплуатировавшиеся в то время:

- YellowPages – содержала данных об организациях-партнерах, а также их участиях в разнообразных долгосрочных *проектах*, с указанием контактных данные ответственных сотрудников и технических служб, соответствующих этим *проектам*. Основными пользователями YellowPages были администраторы (менеджеры), ведущие соответствующие *проекты*; (фактически, проекты могли быть самостоятельными организациями, совместно использующими один и тот же управленческий и технический персонал). Технически система представляла собой ненормализованную реляционную БД под управлением MySQL 3.x (http://www.mysql.com), доступ к которой осуществлялся через набор CGI-скриптов на языке perl (web-приложение).
- LINKS – содержала данные об сетевом и коммуникационном оборудовании (маршрутизаторах, коммутаторах, модемах и т.п.); коммуникационных каналах, к нему подключенных; а также – контактные данные ответственных технических лиц и служб. Потребителем этой информации являлась круглосуточная дежурная служба мониторинга, а поставщиком – техническая служба LINKS, ответственная за физическое обслуживание каналов и оборудования. Проблемой этой БД была плохая заполняемость и недостоверность большинства карточек, информация в которые должна была вручную переноситься из YellowPages. Технически LINKS базировалась на движке системы Whois, разработки RIPE (http://ripe.net/), хранилищем информации в котором являлись хэшированные файлы, а пользовательский интерфейс обеспечивался web-приложением, состоящим из CGI-скриптов на perl.

От новой ИС требовалось обеспечить полную замену этих двух систем, дополнив административную подсистему учетом договоров и соглашений с организациями-партнерами, а также обеспечив возможность привязки технических объектов к договорам. Последнее автоматически гарантировало доступность техническим службам актуальной информации, вводимой администраторами, без ее ручного дублирования. Название новой системы было образовано от имен ее предшественников: YellowPages+LINKS – YellowLinks – YelL – YELL.

Мое участие в проекте началось с изучения и документирования информационных потребностей служб мониторинга и LINKS, которые должны были быть учтены в новой ИС; продолжилось участием в проектировании структуры БД, перешедшем в проектирование и реализацию системы разграничения доступа, основанной на ведении информации через комплекс хранимых процедур (ядро); а после завершения «ядра системы» – перешло в разработку пользовательского интерфейса к СРД и подсистеме LINKS; финалом стало написание технической документации, описывающей архитектуру системы, и «Руководства пользователя» по подсистемами СРД и LINKS.

Базовые средства были выбраны еще до моего вступления в проект, а их окончательные состав таков – ОС FreeBSD 4.11; СУБД ORACLE™ 8.0.5; web-сервер Jakarta Tomcat 4.x, для исполнения web-приложений; Apache 1.3.x+mod_ssl для шифрования трафика; Java (JDK 1.3) для написания web-приложений. «Ядро системы» было написано на ORACLE SQL и PL/SQL.

Над проектом, в режиме полной занятости, работали 2 программиста: Семин Максим Анатольевич и я. Руководил нашей командой начальник группы разработчиков – Всеволодов Михаил Меркурьевич, а заказчиком и главным административным движителем разработки и внедрения являлась Воронина Елена Павловна, директор АНО "ЦВКС "МСК-IX". В окончательном варианте система YELL обслуживает внутренние потребности следующих организаций и проектов: РосНИИРОС (http://ripn.net); MSK-IX (http://www.msk-ix.ru/); RELARN (http://relarn.ru); RBNET (http://rbnet.ru).

Общий объем программного кода системы по состоянию на апрель 2005-го – чуть более 70-ти тысяч строк (не считая 3-х тысяч миграционного кода); документации – 350 страниц формата A4. Количество типов документов (*объектов данных*) – 5; количество подсистем – 3, одна системная и две прикладных, соответствующих двум замещаемым ИС. Первая полнофункциональная бета-версия, объемом 54 тыс. строк (77% от 70-ти), была получена осенью 2002-го, через 9 месяцев после начала кодирования; еще через 3 месяца система достигла 60 тыс. строк (85%); эксплуатация системы начата через 15 месяцев после начала кодирования, в мае 2003-го (объем кода 63 тыс. строк или 90% от окончательного); оставшиеся 10% «веса» система набирала в течении 2-х лет, в процессе доводки и реализации новых потребностей; параллельно доводке велась работа над технической документацией, завершенная в октябре 2004-го, и окончательной версией «руководства пользователя», завершенная к апрелю 2005-го.

Пользовательский интерфейс системы был реализован в виде web-приложений, функционирующих под управлением сервера Jakarta Tomcat и взаимодействующих с ORCALE в рамках архитектуры клиент-сервер. Ввод и корректировка информации в БД осуществлялись только через комплекс хранимых процедур (с применением понятия «*логическая транзакция*»), а чтение – только через фильтрующие view, что сохраняло все достоинства и функциональные возможности реляционного подхода для использования в прикладных приложениях. Ключевым инструментом разработки и развития системы стал кодогенератор, позволивший выполнять любую модификацию структуры БД, и вытекающие из нее изменения хранимых процедур и view в течении одного-двух дней. В «*ядре системы*» были реализованы и использованы концепции «*объект данных*» и «*область данных*», а СРД в своей работе использовала только эти абстрактные понятия (и соответствующие им служебные поля стандартного заголовка), инвариантные к терминологии прикладной области. Для YELL были проработаны принципы разграничения доступа на уровне типов записей и типов полей в пределах области данных (в дополнение к разграничению на уровне типов *объектов данных* в *области данных*). Идеи *индексных таблиц*, и *полей-свойств*, также возникла во этой системе.

На момент написания этого документа, YELL является наиболее близкой к TIS/SQL, реально-эксплуатируемой системой. Однако существующие расхождения между моими авторскими и имущественными правами на ее программный код и техническую документацию потребовали, при выполнении описания принципов TIS/SQL подхода, дистанцироваться от нее, и многое начать с нуля.

2006 – 2008 TIS/SQL подход (http://mave.ru)

Моя первая, неудачная, попытка сформулировать концепцию разработки объектных БД на основе реляционных СУБД относится к 1998 году, и была реакцией на некоторые проблемы проекта KI-MACS. Дальше первых черновиков эта работа не продвинулась, но уже в них было заявлено два базовых термина object (*объект*) и scope (*область*). Для понятия object предпринималась попытка провести аналогию с понятием файл, из которого выводилось требование аналогичной файлу парадигмы манипулирования: открыть/записать/закрыть. Понятие scope вводилось как абстрактный эквивалент прикладному понятию KI-MACS «Ядерная Установка» (Facility), задействованному системой разграничении доступа в качестве критерия деления данных на массивы с различной политикой.

Второй подход к этой теме был предпринят в 2001 году, в момент выхода из проекта KI-MACS. Был составлен план необходимых работ, включавший написание книги с рабочим названием «Проектирование и реализация объектных БД на базе современных, пост-реляционных СУБД», ориентировочного объема 300 страниц A4, и референсной информационной системы, демонстрирующей изложенные в ней принципы. Далее наступил спад, возникли более приоритетные задачи - написание «KI-MACS FrontEnd. Руководство программиста», которое несколько остудило графоманский жар, а учеба и поиски новой работы отодвинули осуществление всех прочих планов на неопределенное будущее.

Впрочем новая работа, в течении нескольких месяцев, привела меня в проект, результатом которого стала система YELL, воплотившая 90% идей несостоявшейся книги. Но здесь возникло неожиданное препятствие – скользкая проблема имущественных прав на код и документацию,

написанные мной, о которых мне начали напоминать работодатели. С конца 2003-го перспективы дальнейшего развития системы стали неопределенными, и я занялся ее документированием, планируя по завершению стать безработным, чтобы спокойно вернуться к исходному плану «книга+референсная система». Однако, чем дальше, тем яснее становилось – все что хотел и мог написать, я уже написал или напишу, завершая документацию YELL, но все это мне как бы не принадлежит, при том, что останется непрочитанным и невостребованным, а мне придется переписать все с нуля. Результатом стала глубочайшая депрессия, и разочарование.

Доводка YELL и работа над документацией затянулись до начала 2005 года, а душевный разлад до конца 2006-го. Параллельно я многократно начинал и бросал работу над новой, независимой реализацией принципов TIS/SQL подхода, одновременно пытаясь обрести душевное равновесие с помощью разнообразных гуманитарных наук и занятий (экономика, история, философия, филология, лингвистика, искусствоведение, фотография, музыка, архитектура, литература и поэзия). И все это, в конце-концов, оказало свое опосредованное влияние на TIS/SQL подход.

Новый этап начался в конце 2006-го года, с кардинального пересмотра первоначального плана. Во главу угла был поставлен принцип «меньше – значит лучше, значит меньше – значит больше», из которого следовало:

- Не стоит писать 300 страниц, которые никто не станет читать, поскольку современный ритм жизни не располагает к подобному добровольному чтению – лучше написать 30 страниц, которые с гораздо большей вероятностью будут прочитаны, хотя-бы из любопытства.
- Не надо создавать масштабную и «совершенную» референсную систему в 100 тысяч строк, тем более в одиночку. Такие системы выглядят пугающе и способны скорее дискредитировать предлагаемые идеи, чем привлечь к ним сторонников. Системы с недостатками вызывают гораздо больше дискуссий и желания что-то в них доделать, что для продвижения новых идей – «самый главный витамин».

Результатом стала работа, чье сокращенное название звучит как «TIS/SQL подход. Манифест разработчика». Уложить весь «манифест» в 30 страниц не удалось, но первые две главы, содержащие все идеи и понятия, удалось вместить в 40 страниц, что вполне приемлемый объем для заинтересованного читателя. Оставшиеся 40-50 страниц – составляют более конкретные рекомендации по реализации «ядра системы» и разнообразных приложений, его использующих.

В дополнение к вопросам, проработанным при работе над YELL, TIS/SQL подход содержит проработку вопросов реализации и использования мандатной СРД; журнала аудита; контейнеризации и перемещения ОД; построения распределенных, реплицируемых систем. Остальные изменения носят косметический характер.

Главной задачей «манифеста» стало отделение общих принципов и проблем, достойных внимания потенциального читателя, от интересных, но частных потребностей (и соответствующих им решений) конкретных систем-предшественников. Главной психологический проблемой, тормозившей работу, стало – отсутствие в обозримой перспективе возможностей для реального применения результатов проделанной работы, поскольку «манифест», по определению, не должен стать надгробной плитой с эпитафией, но ступенью, с которой становятся видны новые горизонты.

Евстропов А.В. апрель 2008

Приложение B Пример реализации простейшей TIS (PostgreSQL 8.x)

```
-- 1. Создать пользователей и БД (выполняет администратор СУБД)
CREATE USER tis;
CREATE USER tisuser1;
CREATE USER tisuser2;
CREATE DATABASE tisexmpldb OWNER tis ENCODING 'KOI8-R';

-- 1.1. Выполнить в БД tisexmpldb от имени пользователя pgsql,
-- чтобы запретить ординарным пользователям создавать таблицы
-- и функции, доступные всем остальным пользователям
-- REVOKE CREATE ON SCHEMA public FROM PUBLIC;
-- REVOKE USAGE ON SCHEMA public FROM PUBLIC;

-- 2. Инициализация БД (выполняет владелец БД, пользователь tis)
CREATE SCHEMA tis;
GRANT USAGE ON SCHEMA tis TO PUBLIC;
CREATE TABLE tis.SAMUsers ( id int, db_user name, name char(16),
                        CONSTRAINT SAMUsers_pkey PRIMARY KEY (id) );
CREATE UNIQUE INDEX SAMUsers_idx_db_user on tis.SAMUsers(db_user);
CREATE TABLE tis.SAMACL ( user_id int, scope_id int,
                      read_allowed char(1), write_allowed char(1),
                      CONSTRAINT SAMACL_pkey PRIMARY KEY (user_id, scope_id) );
CREATE FUNCTION tis.sam_get_user() RETURNS integer STABLE
  AS 'select id from tis.SAMUsers where db_user = session_user'
  LANGUAGE SQL SECURITY DEFINER;

-- 3. Инициализация СРД
INSERT into tis.SAMUsers values(1,'tisuser1','user1');
 INSERT into tis.SAMACL values(1,1,'Y','Y');
 INSERT into tis.SAMACL values(1,2,'Y','N');
INSERT into tis.SAMUsers values(2,'tisuser2','user2');
 INSERT into tis.SAMACL values(2,2,'Y','Y');
 INSERT into tis.SAMACL values(2,1,'Y','N');
INSERT into tis.SAMUsers values(3,'tis','user3');
 INSERT into tis.SAMACL values(3,3,'Y','Y');
-- 4. таблица и view для прикладных данных
CREATE TABLE tis.Somedata (id int not null, scope_id int not null,
                           user_id int not null, value varchar(32),
                          CONSTRAINT Somedata_pkey PRIMARY KEY (id));
CREATE SEQUENCE tis.seq_Somedata;
CREATE VIEW tis.V_Somedata as select * from tis.Somedata where exists
  (select user_id from tis.SAMACL where user_id = tis.sam_get_user() and
  Somedata.scope_id = scope_id and read_allowed = 'Y');
GRANT SELECT ON tis.V_Somedata TO PUBLIC;

-- 5. Создание процедуры для манипулирования данными
-- 5.1 Тип данных для хранения результата исполнения функции

CREATE TYPE retstatus AS (id int, errcode int, errdesc varchar(255));

-- 5.2 Функция для добавления записи Somedata
CREATE OR REPLACE FUNCTION create_Somedata(scope_id int, value varchar(32))
  RETURNS retstatus LANGUAGE plpgSQL SECURITY DEFINER as $$
```

```
DECLARE
 rets retstatus;
 v tis.Somedata%ROWTYPE;
 i int;
BEGIN
 v.scope_id := scope_id; v.value = value;
 -- 1) Проверить уровень изоляции транзакции
 if pg_catalog.current_setting('transaction_isolation') <> 'read committed' then
    rets.id:=null; rets.errcode:=1;
    rets.errdesc:='Требуется уровень изоляции READ COMMITTED';
    RETURN rets ;
 end if;
 -- 2) Идентифицировать пользователя
 v.user_id := tis.sam_get_user();
 if v.user_id is null then
    rets.id:=null; rets.errcode:=2; rets.errdesc:='Пользователь не определен';
    RETURN rets ;
 end if;
 -- 3) Проверить права доступа
 SELECT INTO i count(*) FROM tis.SAMACL acl where acl.user_id = v.user_id
    and acl.scope_id = v.scope_id and acl.write_allowed='Y';
 if i = 0 then
    rets.id:=null; rets.errcode:=3; rets.errdesc:='Отказ в доступе';
    RETURN rets ;
 end if;
 -- Выполнить добавление записи
 v.id=nextval('seq_Somedata');
 INSERT INTO tis.Somedata values(v.id,v.scope_id,v.user_id,v.value);
 -- Вернуться
 rets.id:=v.id; rets.errcode:=0; rets.errdesc:='Сделано';
 RETURN rets ;
END $$;
GRANT EXECUTE ON FUNCTION tis.create_Somedata(int,varchar(32)) TO PUBLIC;
-- 6. Тестирование - добавление данных различными пользователями
-- 6.1. Выполнить пользователем tis
select tis.create_Somedata(3,'Запись в области 3');
select tis.create_Somedata(2,'Запись в неразрешенной области 2');
select * from tis.V_Somedata;

-- 6.2. Выполнить пользователем tisuser1
select tis.create_Somedata(1,'Запись в области 1');
select tis.create_Somedata(2,'Запись в области 2');
select * from tis.V_Somedata;

-- 6.2. Выполнить пользователем tisuser2
select tis.create_Somedata(1,'Запись в области 1');
select tis.create_Somedata(2,'Запись в области 2');
select * from tis.V_Somedata;

-- 6.3. И в завершение - снова пользователем tis
select * from tis.V_Somedata;
select * from tis.Somedata;
-- END OF EXAMPLES
```

Результаты выполнения тестовых запросов тремя различными пользователями

```
tisexmpldb=> -- 6. Тестирование - добавление данных различными пользователями
tisexmpldb=> -- 6.1. Выполнить пользователем tis
tisexmpldb=> select tis.create_Somedata(3,'Запись в области 3');
 create_somedata
-----------------
 (1,0,Сделано)
(1 запись)

tisexmpldb=> select tis.create_Somedata(2,'Запись в неразрешенной области 2');
    create_somedata
------------------------
 (,3,"Отказ в доступе")
(1 запись)

tisexmpldb=> select * from tis.V_Somedata;
 id | scope_id | user_id |       value
----+----------+---------+--------------------
  1 |        3 |       3 | Запись в области 3
(1 запись)

tisexmpldb=> -- 6.2. Выполнить пользователем tisuser1
tisexmpldb=> select tis.create_Somedata(1,'Запись в области 1');
 create_somedata
-----------------
 (2,0,Сделано)
(1 запись)

tisexmpldb=> select tis.create_Somedata(2,'Запись в области 2');
    create_somedata
------------------------
 (,3,"Отказ в доступе")
(1 запись)

tisexmpldb=> select * from tis.V_Somedata;
 id | scope_id | user_id |       value
----+----------+---------+--------------------
  2 |        1 |       1 | Запись в области 1
(1 запись)

tisexmpldb=> -- 6.2. Выполнить пользователем tisuser2
tisexmpldb=> select tis.create_Somedata(1,'Запись в области 1');
    create_somedata
------------------------
 (,3,"Отказ в доступе")
(1 запись)

tisexmpldb=> select tis.create_Somedata(2,'Запись в области 2');
 create_somedata
-----------------
 (3,0,Сделано)
(1 запись)
```

```
tisexmpldb=> select * from tis.V_Somedata;
 id | scope_id | user_id |       value
----+----------+---------+--------------------
  2 |        1 |       1 | Запись в области 1
  3 |        2 |       2 | Запись в области 2
(записей: 2)

tisexmpldb=> -- 6.3. И в завершение - снова пользователем tis
tisexmpldb=> select * from tis.V_Somedata;
 id | scope_id | user_id |       value
----+----------+---------+--------------------
  1 |        3 |       3 | Запись в области 3
(1 запись)

tisexmpldb=> select * from tis.Somedata;
 id | scope_id | user_id |       value
----+----------+---------+--------------------
  1 |        3 |       3 | Запись в области 3
  2 |        1 |       1 | Запись в области 1
  3 |        2 |       2 | Запись в области 2
```

Приложение C Глоссарий

COM – Component Object Model (модель компонентных объектов). Разработанная Microsoft технология создания программного обеспечения на основе динамически подключаемых и распозноваемых объектных компонентов-серверов, предоставляющих ограниченный доступ к функциональности своих объектов в рамках заранее определенных интерфейсов. Компоненты-сервера могут находиться в одном процессе с их потребителем, или в разных. В последнем случае, один и тот же объект может использоваться нескольким потребителями одновременно. Технология COM является платформо-независимым, простым и элегантным воплощением идей полиморфизами и инкапсуляции ООП, лежащим в основе множества других технолой, таких как OLE, OLE Automation, ActiveX, DCOM, COM+.

CORBA – Common Object Request Broker Architecture (общая архитектура брокера объектных запросов). Определенный и продвигаемый консорциомом OMG (http://www.omg.org) стандарт написания распределенных приложений, нейтральный к используемым языкам программирования и операционным системам.

DCOM – Distributed Component Object Model.

deadlock – безвыходная ситуация; ситуация взаимоблокирования; в управлении базами данных – ситуация, когда несколько транзакций не могут завершиться, поскольку каждая из них ожидает освобождения каких-то ресурсов, занятых другой ожидающей транзакцией.

EJB – Enterprise Java Beans. Спецификация технологии разработки и эксплуатации серверных компонент, содержащих бизнес-логику, и исполняемых под управлением сервера приложений.

ERP – Enterprise Resource Planning планирование и управление ресурсами предприятия (необходимыми для осуществления продаж, закупок и учета при выполнении заказов клиентов в сферах производства, дистрибуции и оказания услуг; о методологии и системе)

IPC – InterProcess Communication. Общее обозначение для любых механизмов взаимодействия процессов, как в пределах одной машины, так и через сеть.

KI-MACS – Kurchatov Institute Material Accounting and Control System. Система учета и контроля ЯМ, разрабатываемая в РНЦ «Курчатовский Институт» с 1994 года.

RMI – [Java] Remote Method Invocation. Средство для создания Java объектов, допускающих вызов своих методов из другой JVM, в т.ч. по сети. Аналог протокола RPC, используемый в распределённых объектных Java -приложениях.

RPC – Remote Procedure Call. удаленный вызов процедур (средство передачи сообщений, которое позволяет распределенному приложению вызывать сервис различных компьютеров в сети; обеспечивает процедурно-ориентированный подход в работе с сетью; применяется в распределенных объектных технологиях, таких как, DCOM, CORBA, Java RMI)
SAM – Security Access Management. В TIS/SQL подходе – синоним СРД
SQL – Structured Query Language; язык структурированных запросов;
TCB – Trusted Computer Base (TCB) – совокупность программного и аппаратного обеспечения, являющаяся критичной с точки зрения защиты информации (как от несанкционированного доступа, так и от нарушения целостности)
UTC – «Universal Time, Coordinated». Всеобщее Скоординированное Время. (Соответствует реальному времени по Гринвичу, без переходов на летнее и зимнее)
view – «Представление». Объект реляционной БД, поддерживающей язык SQL. Является предопределенным запросом на языке SQL, которые пользователи могут использовать в других SQL запросах как обычную реляционную таблицу.
ГАБД – Группа Администратора Баз Данных
ГТК – Государственная Техническая Комиссия при Президенте РФ. (Гостехкомиссия России).
ИС – Информационная Система
МИС – Мета-ядро Информационных Систем.
ОД – Объект Данных (Data Object)
ОИС – Отдел Информационных Систем
ООП – Объектно-ориентированное программирование.
ПО – Программное Обеспечение
ППО – Прикладное Программное Обеспечение
СРД – Система Разграниченя Доступа.
ТЗ – «Технические Задание». Документ, регламентирующий требования к проектируемому изделию.
ЯМ – «Ядерный материал».

Приложение D Алфавитный указатель

Приложение E Список таблиц и иллюстраций

Приложение F Литература и другие источники

Общедоступные печатные источники

P1. К. Дж. Дейт "Введение в системы баз данных" шестое издание. ДИАЛЕКТИКА Киев. Москва 1998.

P2. С.Н. Смирнов, И.С. Задворьев "Работаем с ORACLE". Москва. "Гелиос АРВ" 2002

P3. Майкл Морган "Java 2 Руководство Разработчика". Москва. Санкт-Петербург. Киев. Издательский дом "Вильямс" 2000

P4. "TRUSTED COMPUTER SYSTEM EVALUATION CRITERIA" DEPARTMENT OF DEFENSE (USA) 15 August 1983

P5. Фредерик Брукс. «Мифический человеко-месяц или как создаются программные системы.» Перевод С. Маккавеева. Издательство Символ-Плюс. 2000. Original English language edition Copyright © Addison-Wesley Longman, Inc. 1995

P6. Сингх Лэйв, Келлий Лей, Джо Сафьян и др. «ORACLE 7.3. Пособие разработчика». Издательство «ДиаСофт». Киев. 1997. 1998.

P7. Вильям Дж. Пейдж и др. «Использование Oracle8™/8i™. Специальное издание». Издательский дом «Вильямс». Москва. 2000.

P8. Кент Бек. «Экстремальное программирование». ЗАО Издательский дом «Питер». Санкт-Петербург. 2002.

P9. Эдвард Йордон. «Путь камикадзе. Как разработчику программного обеспечения выжить в безнадежном проекте». Издательство «ЛОРИ». Москва. 2000. 2001.

P10. В.И. Ярочки «Безопасность информационный систем». Москва. «Ось-89». 1996

P11. Дэвид Чеппел «Технологии ActiveX и OLE». Москва. Издательский отдел «Русская Редакция» ТОО «Channel Trading Ltd.», 1997.

P12. Дирк Слама, Джейсон Гарбис, Перри Рассел «Корпоративные системы на основе CORBA». Издательский дом «Вильямс». Москва. 2000.

P13. Майкл Морган «Java 2. Руководство руководство разработчика». Издательский дом «Вильямс». Москва. 2000.

Общедоступные Internt источники

I1. http://wikipedia.org (Интернет энциклопедия)

I2. http://www.citforum.ru/ (Общедоступный массив информации по IT)

I3. http://www.postgresql.org/ (СУБД PostgreSQL, официальный сайт проекта)

I4. http://www.firebirdsql.org/ (СУБД FireBird, официальный сайт проекта)

I5. http://hsqldb.org/ (СУБД HSQLDB, официальный сайт проекта)

Источники ограниченного доступа и рукописи

R1. Исходные тексты и документы проекта KI-MACS. 1994-2001. РНЦ "Курчатов кий институт", НТК-Электроника Отдел Информационных Систем

R2. Исходные тексты и документы проекта YELL. 2001-2005. РосНИИРОС.

R3. «KI-MACS FrontEnd. Руководство программиста.» 2001. Евстропов А.В.

Публикации по технологии МИС

M1. Модульная информационная система. В сб. «Доклады 5-го международного совещания по проблемам математического моделирования, программирования и математическим методам решения физических задач, 20-23 сент. 1983, ОИЯИ, г Дубна», Румянцев А.Н., Малашинин И.И.

M2. Мета-ядро информационных систем МИС-85. Отраслевой фонд алгоритмов и программ ЦНИИАИ Государственного комитета по использованию атомной энергии СССР. Рег №5142/ОФАП 1986 г. Румянцев А.Н., Чижик В.С., Карпов В.В., Васильев А.Л.

M3. Мета-ядро информационных систем МИС-86. Отраслевой фонд алгоритмов и программ ЦНИИАИ Государственного комитета по использованию атомной энергии СССР. Рег №5549/ОФАП 1986 г. Румянцев А.Н., Чижик В.С., Карпов В.В., Васильев А.Л., Михайлюк Д.М., Аленов Ю.В., Остроумов Ю.А.

M4. Система ведения интегрированных баз данных МИС. Препринт ИАЭ – 4496/16-М.:ЦНИИатоминформ, 1987 г. Румянцев А.Н.,Аленов Ю.В., Васильев А.Л., Васильев В.И., Карпов В.В., Михайлюк Д.М., Остроумов Ю.А.,Чижик В.С.

