В декабре 2009 года в Минске компания ЛюксСофт анонсировала начало работ по созданию новой платформы разработки бизнес-приложений – LS Fusion (LSF) О предпосылках возникновения, основных концепциях и возможностях разрабатываемой платформы LS Fusion (LSF) рассказал Алексей Кирковский, руководитель проектов НТООО «ЛюксСофт»

*Сокращения: ООП – объектно-ориентированная парадигма, БД – база данных, БЛ – бизнес-логика, БП – бизнес-процессы

Существующие подходы к разработке бизнес-приложений.

В настоящее время основным стандартом для построения любых информационных систем де-факто является объектно-ориентированная парадигма. Об этом говорит хотя бы то, что количество прикладных языков программирования, реализующих ее, является наибольшим по отношению ко всем другим парадигмам. Основным действующим понятием в информационной системе, построенной на основе объектно-ориентированной парадигмы, является (как нетрудно догадаться) *объект*. Объекты могут создаваться, уничтожаться, изменять свое состояние внутри определенного контейнера. Часто этот контейнер называют *виртуальной машиной*.

На данный момент, у всех реализованных виртуальных машин существует две основные проблемы. Во-первых, виртуальная машина существует только в рамках сеанса операционной системы. Проблема заключается в том, что при завершении сеанса (или при аппаратном сбое), виртуальная машина тоже закрывается, вместе с собой уничтожая всю информацию об объектах, которые в ней находились. Во-вторых, скорость обработки информации у виртуальной машины относительно медленная. Даже простое суммирование значений свойства объектов одного класса может занять достаточно продолжительное время, в случае если объектов относительно много. Для решения этих двух проблем, разработчикам приходится использовать *системы управления базами данных* (СУБД).

СУБД предоставляют и достаточно удобные средства для сохранения состояний объектов, и высокую производительность для обработки информации об объектах. Однако это, в свою очередь, создает новые проблемы. Во-первых, разработчику приходится везде самому заботиться о синхронизации объектов в виртуальной машине и базе данных, тем самым, выполняя много рутинной работы. Вторая проблема - так называемый *семантический разрыв*. Дело в том, что разработчику приходится использовать две парадигмы – объектно-ориентированную для самой информационной системы и реляционную для базы данных. Объектно-ориентированная парадигма будет оперировать объектами, классами, интерфейсами, свойствами, методами. А реляционная – таблицами, полями, индексами, тригеррами, SQL-запросами, ограничениями constraints). Таким образом, часть единой логики системы будет находиться в одной парадигме, а часть в другой, что уже само по себе выглядит «криво». Со временем, для решения этих проблем, начали появляться новые технологии. Большинство из них пытаются просто избавить разработчиков от технической рутины, предоставляя механизмы для автоматической синхронизации объектов в информационной системе и базе данных. Например, Hibernate, Spring, EJB, ADO.NET. В свою очередь производители баз данных тоже начали делать шаги навстречу, создавая так называемые объектно-реляционные (Oracle, MSSQL2005, PostgreSQL) или объектно-ориентированные базы данных (IBM Lotus Notes, Cache и т.д.). Последние, в свою очередь, уже больше являются платформами для разработки приложений, нежели СУБД. Однако, до конца решить проблему им не удается хотя бы в силу того, что объектно-ориентированная парадигма значительно отличается от реляционной и невозможно построить прозрачное отображение одной на другую. Таким образом, даже с использованием этих технологий, разработчику приходится уделять много времени для организации взаимодействия между программой и базой данных, что в свою очередь приводит к значительному увеличению стоимости разработки.

Главным субъектом любой информационной системы, несомненно, является пользователь. При разработке приложения необходимо предоставить ему некий механизм взаимодействия с системой. К сожалению, пользователю недостаточно просто консольной строки для общения с приложением (как, например, изначально предполагалось при разработке языка запросов SQL). Он хочет, чтобы все было как можно более красиво и эргономично. Поэтому, как правило, процесс построения пользовательских интерфейсов является одним из самых дорогих процессов во время разработки. В настоящее время существует достаточно большой набор инструментальных средств для построения интерфейсов для работы с обычными многооконными приложениями (rich-client), так и для интерфейсов в виде страничек в браузерах (thin-client).

Большинство таких средств оперирует такими понятиями как формы, поля ввода, таблицы, кнопки и т.д. Это в свою очередь создает еще один *семантический разрыв*. И устранять его приходится именно разработчику. Для этого ему необходимо построить дизайн формы (расположить объекты на форме), а также организовать взаимодействие построенной формы непосредственно с основными объектами логики системы. Все это требует от разработчика выполнять большое количество рутинных операций (особенно, когда в приложении требуется построить достаточно большое количество очень похожих форм и бизнес-процессов). Кроме того, это значительно ограничивает расширяемость системы, так как при значительном изменении основной логики придется переписывать все формы практически с нуля.

Помимо этого, в общепринятых методиках построения информационных систем существует логическое разделение представляемых пользователю данных на отчеты и формы ввода. Предполагается, что для получения данных он должен использовать отчеты, а для ввода – формы ввода. На практике же, пользователь формирует отчеты для принятия какого-то решения (decision support), а результаты этого решение чаще всего ему нужно обратно ввести в информационную систему. В итоге, в существующей парадигме предполагается что алгоритм работы пользователя зачастую должен быть представлен в виде: сформировать какой-то отчет, распечатать его, проанализировать, принять какое-то решение и ввести результаты решения обратно в систему (пример – принятие решения о закупке товара). Такой подход создает массу неудобств пользователю, хотя при этом многие из них этого осознают, что на самом деле, их работу можно значительно оптимизировать. Для этого необходимо максимально интегрировать отчеты в формы ввода, в которые, в конечном итоге, и вводятся результаты принятия решений (именно поэтому, принцип "документ – строки документа" – ущербный). А непосредственно печать, при необходимости вывести для каких-то целей информацию на бумажный носитель, должна происходить непосредственно из самой этой формы.

Рассмотрим еще один сложившийся стереотип при разработке приложений. Как правило, во время разработки и эксплуатации системы существуют четыре действующих лица: бизнес-аналитик, разработчик, информационная система и пользователь. Бизнес-аналитик формирует требования к системе, разработчик их реализует, информационная система и пользователь их выполняет взаимодействия друг с другом. Вся эта схема основывается на том, что задача пользователя просто выполнять заданные ему сверху бизнес-процессы. В большинстве случаев это оправданно. Однако, в случае когда в качестве пользователя информационной системы выступает высококвалифицированный топ-менеджер (или просто менеджер звена уровня выше среднего), который имеет право изменять бизнес-процессы и, по сути, выступать в роли бизнес-аналитика, то возникает проблема. Ему приходится формулировать все требования по изменению системы бизнес-аналитику, тот в свою очередь дает указания разработчику и тот модифицирует систему. Все это создает дополнительные затраты времени, денег, и придает большую инертность всем бизнес решениям. В идеале, пользователь должен максимально сам "говорить" информационной системе о том, что он хочет изменить в ней или получить от нее, не привлекая при этом разработчика. Тем самым, значительно ускорятся и удешевятся все процессы, связанные с расширяемостью приложения. (пример – формирование какого-то отчета в LS Trade)

Обобщая все вышесказанное можно сказать что, в большинстве текущих информационных систем можно выделить три основных блока: СУБД, основная логика информационной системы (бизнес-логика) и пользовательский интерфейс. Такая модель построения систем носит название 3-х уровневая архитектура приложения (3-tier application). С одной стороны такая модель высокотехнологична (например, позволяет строить кластеры отказоустойчивых серверов, выполняющих тот или иной блок). Кроме того, позволяет разбить архитектуру системы на несколько слабосвязанных компонент, что помогает распараллелить процесс разработки. Однако, в то же время, такая модель создает достаточно большие семантические разрывы между своими блоками и требует от разработчика реализовывать множество рутинных, технических операций для устранения этого семантического разрыва, что значительно удорожает разработку. Если посмотреть на реализацию любой из таких систем, то в ней можно легко обнаружить большое количество классов выполняющих, по сути, техническую (бюрократическую) функцию простого перекидывания запросов друг другу, не выполняя при этом никаких значимых логических функций. Одна из целей построения платформы заключается в том, чтобы система взяла на себя реализацию всех технических аспектов системы, дав возможность разработчику сконцентрироваться непосредственно на основной логике системы. При этом физическая модель все равно остается 3х уровневой. Изменяется лишь логическая структура. Так, уровень СУБД становится абсолютно прозрачным для разработчика, а уровень пользовательского интерфейса максимально приближается к уровню основной логики системы. В итоге, получившееся решение наследует все основные преимущества 3-х уровневой архитектуры – высокую технологичность и

возможность разделить процесс разработки логики и пользовательского интерфейса, при этом значительно уменьшая все семантические разрывы между блоками.

Таким образом, были сформулированы основные проблемы, возникающие на этапе разработки и эксплуатации информационных систем. Имея большой опыт разработки разнообразных приложений, каждый день нам приходилось (и приходится до сих пор) сталкиваться с ними. Цель создания платформы заключается в попытке решить в той или иной степени каждую из этих проблем один раз и навсегда.

Разрыв между БД и БЛ

Начнем с устранения главного 1-го разрыва между БД и БЛ. Как уже упоминалось, основной проблемой является различие между ОО парадигмой и парадигмой SQL. Уже тот факт, что они тесно используются для решения одной и той же задачи говорит о несовершенстве каждой из них. Поэтому единственным вариантом устранения такого разрыва является использование новой обобщенной парадигмы, которая будет обладать свойствами их обоих. Для этого подчеркнем недостатки, которыми обладает классическая ООП и которые необходимо устранить.

Множественные свойства

Первой проблемой классической объектно-ориентированной парадигмы является ограничение принадлежности свойства ровно одному объекту. На практике свойства зачастую могут отражать характеристики нескольких объектов. Так например можно говорить о свойствах остаток на складе – которое является атрибутом пары склад\товар, или договор музыкальной группы с звукозаписывающей компанией – которое также действует именно для связки группа\компания. Для реализации таких связей в ООП, как правило, один из объектов определяется "главным", а остальные представляются в виде ключа в дополнительном классе Map, или же в случае метода как один из параметров. Но очевидно, что такой подход не отражает "истинную" семантику свойств, что приводит к уменьшению расширяемости получаемых решений. А также если учесть, что, например, парадигма SQL косвенно подразумевает множественные свойства, это является одной из причин неизбежности семантического разрыва между БЛ и БД. В обобщенной же парадигме свойство изначально на вход принимает несколько объектов, что по сути выносит именно его, а не класс на первый план. Поэтому можно говорить что получаемая парадигма скорее свойство, нежели объектно-ориентированная. Кроме устранения описанной проблемы это также дает дополнительное преимущество, которое можно сформулировать как "полиморфизм свойств", и которое сильно повышает расширяемость систем использующих эту парадигму. Как и полиморфизме объектов заключается в том, что у любого свойства можно подменить его реализацию с целью расширения или сужения функционала, при этом работа всех остальных свойств никак не будет затронута. И это можно делать, не изменяя ни классов объектов, ни других смежных свойств.

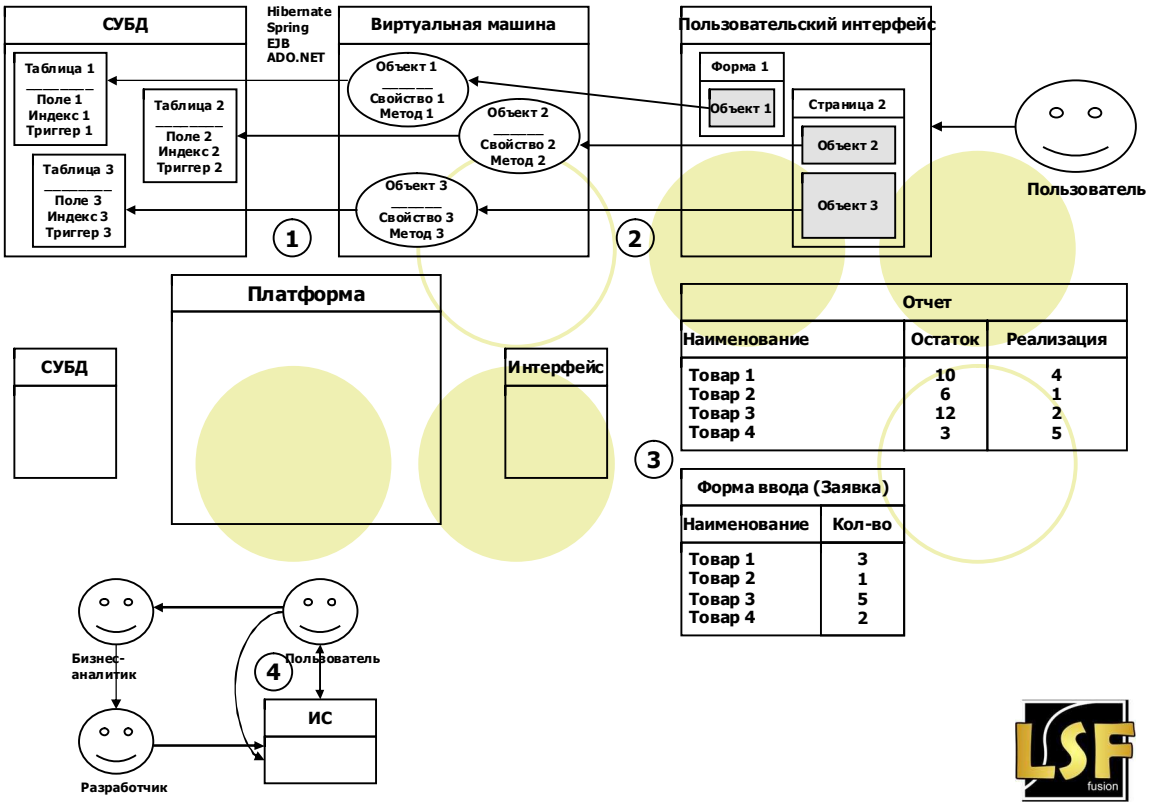
Динамическая классификация

Также одним из ограничений объектно-ориентированной парадигмы является то, что класс объекта известен заранее и не может изменяться. В то время как в жизни свойства объекта существуют только относительно субъекта, который с ним взаимодействует. То есть, например, если у человека в руках появляется некоторый объект то, что с ним можно делать он узнает не сразу, а в процессе его изучения, при этом он может ошибаться. В бизнес-решениях в качестве примера такой ситуации можно рассматривать статусность, когда объект (документ, товар) проходит через несколько "рук", обрастая все новой и новой информацией. Исходя из этого, в обобщенной модели добавление объекта напрямую не привязывается к его классификации. Более того, первая операция рассматривается как частный случай второй. Другими словами с точки зрения системы изначально существует бесконечное множество объектов некоторого самого базового класса (на физическом уровне в виде некоторого пула идентификаторов), а пользователь в процессе работы лишь говорит об изменении их классов.

Свойства и ограничения

Третья проблема сводится к разделению в ООП понятий полей и методов. Это разделение изначально обусловлено требованиями к "физическому" выполнению программы на существующих виртуальных машинах, однако в бизнес-системах отличие между полями и методами, если ассоциировать последние с их значениями, далеко не очевидно. Строго говоря, конечному пользователю или другому объекту системы все равно является ли свойство первичным или рассчитывается по какому-нибудь правилу. Поэтому с точки зрения бизнес-системы такое разделение избыточно, и соответственно это также нашло отражение в обобщенной парадигме.

3х уровневая архитектура приложения



ООП является парадигмой программирования, и реализация ее методов построена на структурном программировании (подпрограммах, ветвлении, циклах и т.д.). В то же время SQL парадигма базируется на реляционной и булевой алгебрах, операциях группировки и сортировки. Соответственно она более высокоуровневая, поэтому немного ограничена, но именно это позволяет ей работать гораздо быстрее, и собственно и привело к появлению уровня СУБД как такового. Естественно чтобы обобщенная парадигма могла достаточно эффективно функционировать, она также должна быть высокоуровневой, однако при этом, позволяя в некоторых случаях "опускаться" на уровень структурного программирования, в том случаю когда бизнес-логика будет алгоритмически очень сложной.

Обобщенная парадигма.

С учетом всех вышеописанных проблем была выработана следующая парадигма разработки. Логика классов и наследования аналогична классической ООП, ее можно представить в виде направленного графа, где наличие ребра означает, что класс, из которого исходит ребро, наследует класс, в который оно входит. Логика свойств строится на основании некоторого базового множество классов свойств. При помощи его можно к существующему множеству свойств, добавлять новые свойства.

(Данная информация является возможным предметом для патентования, поэтому на данном этапе в этом документе не представлена)

Ограничения (Constraints)

На практике учитывать все объекты в мире в бизнес-системе невозможно в силу ряда экономических и технических причин, поэтому очень часто их учитывают вместе. Делают это следующим образом. Рассматривают их самый конкретный класс как объект, а все свойства "преобразуют" во множественные с количественным классом значения и принимающие на один из входов этот объект. Однако при этом, так или иначе, необходимо следить за тем, чтобы не появлялось парадоксальных ситуаций, когда скажем, кол-во пришедших объектов меньше количества ушедших. Для этого в парадигму внесено такое понятие как ограничения (Constraints). Его смысл в том, что на любое свойство в системе можно "повесить" проверку на сравнение (=,>,<) его значения с нулем. При этом этот механизм абсолютно прозрачен для разработчика. Система при выполнении транзакции будет сама следить за тем, чтобы это ограничение не было нарушено. Соответственно в некоторых случаях, когда изменение инициируется не непосредственно пользователем, а репликацией, или же когда пользователь просто не может знать, как разрешить данное ограничение, может возникать вопрос автоматического изменения данных так, чтобы ограничения выполнялись. Самым ярким примером последнего случая может быть "расчет себестоимости", то есть когда расход "расписывается" по приходам. В общем случае задача разрешения ограничений алгоритмически очень сложная, поэтому на начальном этапе развития платформы, просто предусмотрена точка входа для каждого ограничения, где разработчик может дописать свой код по устранению некорректных данных на основе того, на какое кол-во оно превышено, и что именно изменил пользователь.

Физическая модель (оптимизация)

Описанная модель позволяет задать бизнес-логику в некоторой модели, абсолютно абстрагировано от того, как она будет выполняться на "физическом" уровне. Однако перед тем как запустить ее в промышленную эксплуатацию необходимо выполнить некоторую оптимизацию выполнения, которую в силу недостатка статистики и алгоритмической сложности система выполнить не в состоянии. Соответственно "рычагами" которыми может управлять разработчик \ администратор являются:

1. *Таблицы по классам*. Разработчик задает, для каких множеств классов создавать таблицы. Таким образом, для хранения свойства выбирается таблица, множество классов которой, наиболее близко к множеству классов этого свойства. Соответственно чем больше таблиц задано, тем больше будет выполняться операций связывания (Join), с другой стороны тем уже будут записи, для каждого хранимого свойства будет меньше записей со значением null, и общая база будет занимать меньше места.
2. *Постоянно хранимые свойства*. В случае если количество операций чтения свойства существенно превышает количество операций его изменения, для ускорения работы имеет смысл уведомить систему, что такое свойство лучше хранить постоянно, и соответственно изменять сразу при изменении других свойств, влияющих на значение этого свойства. Так, например, рассчитывать остаток или последнюю цену при каждой (достаточно частой) необходимости просто перегрузит сервер базы данных. Включив же эти свойства в список постоянно хранимых, можно, с одной стороны конечно немного увеличив время применения изменений, с другой стороны значительно уменьшить время доступа, тем самым, сбалансировав общую нагрузку.
3. *Индексы*. В том случае если по свойству часто идут операция связывания (Join), группировки (Group) или отбора, для их оптимизации желательно иметь индексы, которые позволят выполнять эти операции не за линейное, а за логарифмическое время. Этот механизм используется только в СУБД (и хорошо там описан), система лишь своего рода "транслирует" список свойств в список полей, из которых потом оформляет индекс, поэтому останавливаться подробно на этом механизме не имеет особого смысла.

Фактически функция оптимизации это больше функция администрирования, нежели разработки. При этом если в классических системах администратор управляет только построением индексов (3-й пункт), то в описанной модели его возможности гораздо шире. Соответственно большинство вопросов скорости работы системы он может и должен решать без привлечения разработчика.

Работа с данными

Как уже упоминалось, в некоторых редких случаях как, например, при разрешении ограничений или в некоторых задачах интеграции разработчику необходимо "опуститься" на более

низкий уровень работы с объектами. Использовать непосредственно SQL в качестве такого уровня неудобно по нескольким причинам:

1. Так или иначе, необходимо использовать основную обобщенную парадигму, так как именно она содержит всю бизнес-логику

2. С точки зрения кросс-платформенности, то есть, чтобы не привязываться к конкретным синтаксисам SQL. Заметим, что большинство существующих систем в этом смысле обладают псевдо-кроссплатформенностью, потому как, возможно в отдельном модуле, но все-таки требуют задания конкретных SQL запросов, что требует их переписывания при переходе на другую СУБД.

3. Из-за несовершенства самой SQL парадигмы. Обычно принято ассоциировать SQL только с реляционной алгеброй, однако, при этом это язык не только связывания, но и отбора данных. Таким образом, он использует и классическую булеву алгебру, однако при этом разбивает ее на 2 части: типы Join'ов (Inner, Left, Right, Full), и собственно сам оператор WHERE, который накладывает условия на непосредственно поля таблиц. Такое разбиение, особенно при использовании SQL на более высоком уровне, во-первых, не удобно в использовании сам по себе, а во-вторых, значительно мешает оптимизации, при которой от Full Join'ов нужно либо вообще избавляться, либо перетаскивать в самый верх запроса. Это необходимо, потому как результат их выполнения, как правило, реализуются в виде функций (COALESCE, NVL,…), а для них SQL оптимизаторы не умеют использовать индексы, что приводит к катастрофическому падению производительности.

Исходя из этого, был разработан отдельный уровень ядра общения с базой данных основанный на обобщенной реляционно-булевой логике. Он же используется и при реализации самой обобщенной парадигмы, что позволяет и ей, в том числе, решать вышеперечисленные проблемы кросс-платформенности и оптимизации запросов.

Распределение \ Зеркалирование БД

Описанный подход также позволяет более прозрачно, нежели SQL серверам решать проблему распределенных баз данных. Эта проблема возникает в том случае, если каналы связи не надежны и имеют тенденцию отказывать в те моменты, когда функционирование базы данных очень критично. Эта проблема как правила перекладывается на SQL сервера, однако в силу того, что они не обладают всей бизнес-логикой (из-за 1-го семантического разрыва), абсолютно прозрачно они ее решить не в состоянии. В то же время обобщенная парадигма имеет единое логическое пространство, поэтому в состоянии автоматически решать проблемы с постоянно хранимыми свойствами и даже ограничениями (в том случае если под них подходят стандартные схемы разрешения или же описаны разработчиком свои).

Пользовательский интерфейс

Главным действующим элементом при взаимодействии пользователя и информационной системы является *форма*. Каждая форма представляет собой, по сути, некоторое окно в систему, позволяющее пользователю через него добавлять, удалять объекты и изменять их состояние. Любая форма в системе состоит из нескольких блоков, каждый из которых отображает объекты одного класса, а также нескольких управляющих кнопок. Каждый из таких блоков в определенный момент времени может находиться в одном из двух состояний: в виде *панели* и в виде *таблицы*. В виде панели в данном блоке виден только один объект, свойства которого отображаются как набор полей для ввода. В виде таблицы в соответствующем блоке видны одновременно несколько объектов. По умолчанию, пользователь может сам переключаться из одного вида в другой. В то же время, разработчик может сам определить, какой вид выбирается по умолчанию, и какой из них разрешен. Для каждого объекта, отображаемого на форме, разработчик обязан задать свойства, которые будут отображаться для него. Кроме того, разработчик может помещать на форму множественные свойства (то есть свойства, которые существуют для нескольких объектов).

По умолчанию, для блока, который находится в виде *таблицы*, будут отображаться все объекты заданного класса. При необходимости можно задать фильтры, применяемые к данным объектам. Каждый фильтр задает свойство, на которое он действует, выражение (=,>,<,>=,<=,<>) и значение, объект или другое свойство. Такие же фильтры может применять и сам пользователь для блока, который находится в виде *таблицы*. Кроме того, разработчик может определить часто используемые отборы и предоставить пользователю возможность выбирать на форме один из них.

У разработчика, как и у пользователя, также имеется возможность задавать порядок, в котором следуют объекты в соответствующем блоке. По умолчанию, данные в таблице выдаются в порядке возрастания идентификаторов соответствующих объектов. Для задания порядка достаточно

указать свойство и направление упорядочивания, по которым должна идти сортировка (для пользователя просто кликнуть по заголовку).

В случае, если у заданного класса для определенного блока существуют потомки (это не листовой класс), то форма автоматически высвечивает соответствующее дерево, предоставляя пользователю интерфейс по отбору объектов только интересующего его класса (у разработчика имеется возможность отключить его при необходимости). На основе текущего выбранного класса (по умолчанию, выбранным считается базовый класс объекта блока) определяется, где должно рисоваться то или иное свойство. Если свойство определено для всех объектов выбранного класса, то оно рисуется в таблице, иначе – в панели.

Кроме того, имеется возможность помещать в один блок сразу несколько объектов. В этом случае в таблице будут присутствовать все пары (или тройки и т.д.) объектов заданных классов. Это очень удобно в случае, когда необходимо работать с множественными свойствами (в частности фильтровать, сортировать и т.д.).

Помимо визуализации данных форма также предоставляет пользователю интерфейс по внесению изменений в информационную систему. Пользователь может добавлять, удалять объекты, а также изменять значения свойств. В силу того, что все свойства и отношения между ними четко формализованы в основной логике, информационная система сама в состоянии определить какие свойства пользователю разрешено изменять, и какие свойства автоматически пересчитывать при модификации каждого конкретного свойства. При редактировании свойства примитивного типа (строка, число, дата) форма сама предоставляет соответствующий объект ввода, в противном случае, пользователю предоставляется диалог по выбору соответствующего объекта. Форма диалога может, как задаваться разработчиком, так и определяться автоматически системой. В случае если пользователь пытается изменить свойство, которое не является первичным, система сама обнаруживает ближайшее свойство, которое ему разрешено модифицировать, и пытается модифицировать его. Таким образом, разработчику вообще не приходится заботиться о редактировании данных в системе.

Все изменения, сделанные в какой-то конкретной форме регистрируются в сессии изменений, привязанной к этой форме. У каждой формы может быть как своя собственная сессия, так и несколько разных форм могут делить одно и ту же. В этом случае все изменения сделанные на одной форме тут же отображаются на другой при переключении. По умолчанию, при вызове диалога, он начинает работать в той же сессии, что и основная форма (пример – изменение наименования товара в диалоге выбора товара и отмена действия в форме внешнего прихода). Таким образом, все изменения, сделанные в рамках работы формы и при вызове диалогов, в базу идут одной единой транзакцией (можно сделать и наоборот).

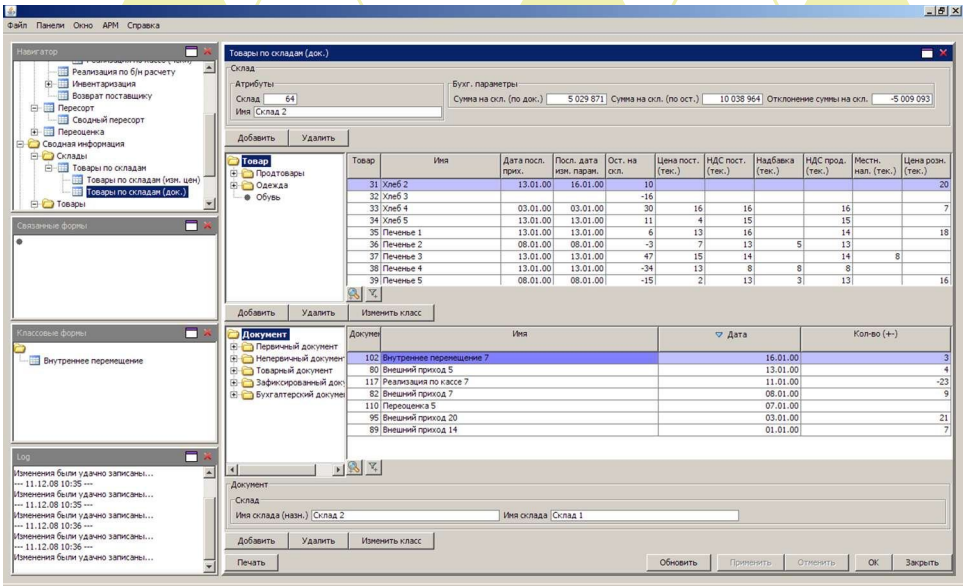
По умолчанию, на всех формах в системе присутствуют четыре основные управляющие кнопки. “Обновить” – перечитывает все данные на форме с учетом изменений сделанных другими пользователями. “Применить” – применяет все изменения, сделанные пользователем в текущем сеансе в основную базу данных. В случае, если при применении изменений нарушаются ограничения, то в лог выводится сообщение о том, какие ограничения нарушены и пользователю дается возможность исправить измененные данные. “Отменить” – отменяет все текущие изменение и получает с сервера старые данные. “Печать” – выводит на печать данные текущей формы с учетом выбранных объектов, примененных фильтров и сортировок (используется LGPL технология JasperReports). Разработчик может сам спроектировать дизайн выходной печатной формы или будет использоваться стандартный сгенерированный системой дизайн. Все объекты, которые на момент нажатия кнопки находятся в виде панели, фиксируются и в выходную форму попадает только текущий объект, а которые находятся в виде таблицы – попадают все объекты в текущем подмножестве. Все печатные формы могут быть экспортированы в различные общепринятые форматы данных (например, в Excel, PDF, JPG) и т.д.

Для задания расположения объектов на форме используется специальный алгоритм, основанный на симплекс методе с использованием метода ветвей и границ. По умолчанию, он старается расположить объекты как можно компактнее, увеличив тем самым размеры всех таблиц на форме. В то же время, разработчику предоставляется возможность манипулировать практически всеми аспектами автоматического расположения. Он может указать пропорции, в которых растягивать объекты, их относительное расположение друг относительно друга, минимальные, предпочитаемые и максимальные размеры. Такой подход необходим для обеспечения расширяемости системы и поддержки гибкой политики безопасности. При разработке формы разработчику не придется концентрироваться на деталях вроде дизайна формы или запрете для показа неавторизированным пользователям какой-либо информации. Ему необходимо лишь задать структуру формы, то есть то, что необходимо видеть пользователю, а всю рутинную работу система возьмет на себя.

Важно отметить, что при таком подходе форма, по сути, выступает в роли браузера, то есть вообще ничего не знает о логике самой системы. Когда пользователь совершает какое-либо действие (переходит с одного объекта на другой, пытается изменить или изменяет какое-либо свойство, жмет любую кнопку и т.д.), форма уведомляет сервер приложений об этом действии, а он, в свою очередь, передает форме все то, что она должна изменить у себя в представлении. При этом интерфейс взаимодействия между ними четко формализован, и передаются лишь те данные, которые видны на форме, предварительно проходя максимальную архивацию, за счет чего достигается ультратонкий клиент (минимальный трафик).

Для управления процессом создания форм в пользовательском интерфейсе существует так называемый *навигатор*. Он состоит из трех видов. *Главный* вид отображает все заданные разработчиком формы в виде иерархического представления. Любая форма, созданная при помощи него, имеет свою собственную сессию изменений. Следующий вид отображает так называемые *связанные формы*, которые могут задаваться разработчиком для каждой конкретной формы. В нем обычно отображаются формы, наиболее часто используемые с заданной. В случае если разработчик указал, что форма является печатной, то при ее открытии через вид *связанные формы*, она сразу будет выведена в печатном виде. Кроме того, все такие формы будут работать с той же сессией изменений, что и форма из которой ее вызвали. Последний, третий, вид работает относительно текущего выбранного объекта на форме и показывает все его *классовые формы*. По умолчанию, система для каждого класса в системе создает свою собственную форму, куда помещает все его базовые свойства и делает ее в качестве классовой для него. Классовые формы предоставляют возможность пользователю быстро и удобно просматривать всю информацию по объекту. Это достигается за счет того, что при создании любой формы система сама делает активными, последние выбранные объекты в рамках всего сеанса работы пользователя. Таким образом, если объект определенного класса присутствует на нескольких формах, то при переключении между ним он будет оставаться одним и тем же. Классовые формы, как и связанные, работают в той же сессии изменений, что и основные формы.

Открытые формы, а также их расположения внутри основного окна, умеют автоматически сохраняться локально, а затем при повторном запуске приложения возвращаться в то состояние, в котором они были во время их закрытия. Таким образом, пользователю предоставляется возможность настроить интерфейс под себя, открыть все то, что ему необходимо и в дальнейшем работать с этим без необходимости делать это каждый раз. Для настройки расположения форм внутри главного окна используется LGPL технология DockingFrames.



Резюме

В основе платформы заложена обобщенная парадигма, позволяющая разрешить 1-й (между логикой системы и базой данных) и в большей степени 4-й (между пользователем и системой) семантический разрывы. С точки зрения технологичности это обеспечивает ей скорость разработки и кросс-платформенность в плане СУБД и серверов приложений. На основе этой же парадигмы разработана концепция автоматического интерфейса, позволяющая устранить 3-й разрыв (между отчетами и формами ввода), пользовательскую часть 4-го, и максимально "сжать" 2-й (между пользовательским интерфейсом и логикой системы).

Преимущества

- Быстрая разработка надежных и высокотехнологичных приложений.
- Ультра тонкий клиент. 3-х уровневая архитектура.
  Единое информационное пространство предприятия.
- Высокая расширяемость и масштабируемость полученных решений.
- Гибкая система безопасности.
- Кросс-платформенность (Linux/Win/MacOS, PostgreSQL/MSSQL/Oracle)

Позиционирование

- Разработка собственных прикладных программных продуктов
- Заказные разработки
- Продажа прав на использование

Наибольшая применимость

- Большое количество относительно сложных бизнес-процессов
- Наличие территориально удаленных объектов
- Необходимость предоставления доступа к системе бизнес-партнерам

Таким образом, платформа удовлетворяет всем современным требованиям предъявляемым к информационным системам, при этом, позволяя разрабатывать приложения гораздо быстрее и дешевле, чем при использовании классических подходов.