# Система DAMP: унифицированный доступ к гетерогенным данным в распределенных приложениях*

С. В. Скударнов
sergey.skudarnov@gmail.com

Проблема унифицированного доступа программных приложений к данным различных типов существует давно. В настоящее время для ее решения имеются такие технологии, как JDO, JAXB, SDO, ADO.NET, а также различные библиотеки, поддерживающие сериализацию (например, MFC). Однако при разработке мобильных и встроенных приложений, систем реального времени и др., где ключевыми факторами являются надежность, производительность, ограниченность вычислительных ресурсов, а также доступность исходных кодов, по-прежнему существует потребность в «легковесных» решениях. В данной статье представлена такая система, называемая DAMP (Data Access Management Pattern). Она включает в себя шаблон проектирования и его реализацию на языке C++. В работе также описана апробация DAMP при разработке сервера мобильных приложений.

## Введение

При разработке систем, которым необходимо интенсивно работать с различными внешними источниками данных (HTML/XML-страницы, RSS-потоки, реляционные базы данных, файловые системы и т. д.), остро встает вопрос о наличии единого инструмента доступа к данным. Ключевыми факторами, характеризующими

* Проект выполнен при поддержке Фонда содействия развитию малых форм предприятий в научно-технической сфере (грант 08-2-Н1.3-0064).

состоятельность такого инструмента, являются единообразие интерфейсов доступа и возможность работать с конкретными видами данных в более абстрактном виде, нежели их изначальное, «сырое» представление.

Проблема унифицированного доступа программных приложений к данным различных типов существует давно. В настоящее время для ее решения имеются такие технологии, как JDO [7], JAXB [6], SDO [5], ADO.NET [2], ориентированные на распределенные Web-приложения, динамическую типизацию и XML-представление данных. Сегодня развитие таких технологий во многом происходит в рамках архитектуры SOA (Service-Oriented Architecture) [3] ввиду активного упрочнения ее позиций в сфере промышленного программирования. Однако все перечисленные выше технологии являются достаточно «тяжеловесными» и ориентированы на использование в больших распределенных бизнес- и Web-приложениях. В частности, для доступа к данным используется динамическая типизация, внутреннее представление данных основано на языке XML и т. п. Это ограничивает область применения таких подходов, поскольку существует большое количество областей, где подобные издержки на производительность недопустимы, т. е. требования к эффективности выполнения и к простоте реализации работы с данными являются критичными. В то же время приложения активно работают с разнородными источниками данных, а доступ к внешним данным осуществляется с более низких архитектурных уровней, чем те, на которые ориентированы указанные выше технологии. Библиотеки, реализующие механизмы сериализации/десериализации (например, библиотека MFC [10]), решают другие задачи — сохранение/восстановление данных (как правило, объектных), передаваемых по сети, а также обеспечение сохраняемости данных для объектно-ориентированных приложений с последующей возможностью их восстановить. Более того, многие из этих библиотек работают только с файловыми источниками данных.

Таким образом, возникает потребность в «легковесных» решениях, которые должны удовлетворять всем или некоторым ключевым факторам из следующего списка: надежность, производительность, ограниченность вычислительных ресурсов, сертифицированность используемых технологий разработки (для стратегических приложений). Такие решения востребованы в мобильных и встроенных приложениях, в системах реального времени и т. д. Например, опыт показывает, что при реализации высокопроизводитель-

ных систем с высокими требованиями по пропускной способности и надежности (например, системы ТВ-вещания) часто приходится переписывать вручную даже драйвера аппаратуры и некоторые низкоуровневые сетевые протоколы.

В данной статье представлена система DAMP (Data Access Management Pattern) — шаблон проектирования (design pattern) для «облегченной» реализации взаимодействия прикладных программ с внешними источниками данных на основе универсального интерфейса. Представлена также реализация шаблона в виде библиотеки классов на языке C++, описана апробация подхода, выполненная в рамках разработки сервера мобильных приложений.

Система DAMP позволяет прикладным программам гибко работать с данными из различных источников, скрывая от этих приложений разнообразные низкоуровневые интерфейсы. Система DAMP реализована на небольшом подмножестве языка программирования C++, ее реализация компактна и проста в освоении и поэтому может использоваться для большого класса систем реального времени и встроенных систем. Центральной абстракцией системы DAMP является класс SessionData — абстрактный тип, к которому преобразуются данные, получаемые из разнообразных источников. При этом структура «сырых» данных статически определяется самим разработчиком средствами языка программирования, а не специальными механизмами динамической типизации, как это делается в технологиях SDO, JDO и т. д. Отказ от динамической типизации и XML-представления данных дает существенный выигрыш в производительности. Таким образом, DAMP предоставляет возможность разработчику приложения контролировать представление «сырых» данных и осуществлять более тонкое взаимодействие с ними. Естественно, унификация накладывает ограничения на использование специфических операций работы с различными типами данных, выходящими за рамки CRUD (Create, Read, Update, Delete), например позиционирование в файле прямого доступа. Однако DAMP оставляет возможность выполнения таких операций посредством платформенно-зависимых механизмов более низкого уровня.

Статья организована следующим образом. Первый раздел посвящен обзору существующих подходов и технологий, близких к тематике статьи. Второй раздел содержит подробное описание архитектуры DAMP и ее составляющих компонентов с примерами реализации на C++. В третьем разделе рассмотрен пример ис-

пользования предлагаемой системы в приложении для мобильных устройств, которое отображает текущие курсы запрашиваемых пользователем валют. В четвертом разделе описываются текущее состояние и статус разработки.

# 1. Обзор существующих подходов и технологий

## 1.1. Технология JDO

Технология JDO (Java Data Objects) [7] была стандартизирована сообществом JCP[1] в 2003 году. JDO представляет собой средство для работы с данными в среде Java из разных источников — баз данных, файловых систем, систем обработки транзакций и т. д. JDO отображает исходные связи между данными в связи между JDO-объектами, предоставляет средства навигации по этим связям, а также обеспечивает параллельный доступ к данным.

## 1.2. Технология EMF

Технология EMF (Eclipse Modeling Framework) [11] предназначена для создания Java-приложений, активно работающих с данными в памяти. Она позволяет генерировать соответствующие Java-классы по XML-схемам, UML-моделям, аннотированным Java-интерфейсам и т. д. Сгенерированные классы для работы с данными имеют следующие функции: сохранение/загрузка, удобный программный интерфейс для работы с данными, валидация модели данных и др. Эта технология интегрирована с Eclipse-средствами разработки графических редакторов, в первую очередь с технологией GMF (Graphical Modeling Framework), и может использоваться для реализации репозиториев в памяти с сохранением и загрузкой данных из XML-формата.

## 1.3. Технология JAXB

Технология JAXB (Java API for XML Data Binding) [6] была выпущена группой JCP в январе 2003 года. JAXB посвящена свя-

[1] JCP (Java Community Process) — открытая организация, состоящая из разработчиков и лицензиатов Java, основана компанией Sun Microsystems и служит для разработки и пересмотра спецификаций технологии Java, эталонных реализаций и комплектов совместимости (оф. сайт: http://www.jcp.org).

зыванию XML-данных — представлению XML-данных в виде Java-объектов в памяти. JAXB освобождает прикладного программиста от необходимости самостоятельного синтаксического разбора или создания XML-документов, выполняет маршаллинг/сериализацию (Java в XML) и обратный процесс (XML в Java).

### 1.4. Технология SDO

Технология SDO (Service Data Objects) [5] изначально являлась совместной разработкой компаний BEA[2] и IBM. Первая версия технологии выпущена в 2004 году. SDO является дальнейшим развитием технологии JDO. В то время как JDO рассматривает только вопрос сохраняемости (persistence issue) (на уровне данных платформы JEE[3] [8] (JEE persistence layer) или на уровне корпоративной информационной системы (EIS) [4]), SDO охватывает более широкий круг вопросов и позволяет работать с данными, которые могут передаваться между любыми уровнями платформы JEE, например, между уровнем представления (presentation layer) и бизнес-уровнем (business logic layer). SDO предоставляет прикладным Java-приложениям унифицированный механизм работы с данными из реляционных баз данных, EJB-компонент, XML-страниц, Web-служб, Java EE Connector Architecture, JSP-страниц и др. SDO-объекты данных хранят свои данные в свойствах, каждое свойство имеет тип, являющийся либо примитивным типом (например, int), либо общеупотребительным типом данных (commonly used data type) (например, Date), либо типом другого SDO-объекта. Каждый SDO-объект предоставляет методы для чтения и записи своих свойств (getters и setters). Предоставляются несколько «перегруженных» версий этих методов, разрешающих доступ к свойствам по их имени (String), по номеру (int) или по собственно мета-объекту свойства. В свою очередь, String-аксессор поддерживает также XPath-синтаксис обращения к свойствам. SDO-объекты связываются вместе и содержатся в специальных графах, которые обес-

[2]BEA Systems (http://www.bea.com) — дочерняя компания Oracle Corporation, специализирующаяся на межплатформенном ПО, предназначенном для осуществления взаимодействия прикладных приложений с базами данных.

[3]JEE (Java Platform, Enterprise Edition) — набор спецификаций и соответствующей документации для языка Java, описывающей архитектуру серверной платформы для задач средних и крупных предприятий. До версии 5.0 она называлась J2EE 1.x (Java 2 Platform, Enterprise Edition, v 1.x). Текущая версия сокращенно называется Java EE 5.

печивают контейнер для дерева SDO-объектов. Графы данных создаются промежуточными службами данных для работы с ними SDO-клиентов. SDO представляет собой изолированную архитектуру, поэтому SDO-клиенты взаимодействуют только с графом данных, они могут просматривать его, читать и изменять его объекты данных. К тому же граф данных может состоять из объектов, представляющих данные из разных источников. Граф данных содержит корневой объект данных, связанные с ним остальные объекты данных и суммарный отчет по изменениям, который используется для фиксации изменений в источнике данных. Предоставляя списки измененных свойств (вместе со старыми значениями), а также списки созданных в графе данных и удаленных из него объектов, суммарные отчеты изменений дают возможность промежуточным службам данных эффективно и последовательно обновлять источники данных.

Фактически SDO полностью включает в себя функциональность технологии JAXB, включая работу с XML-источниками, поддерживая не только статическое, но и динамическое связывание данных. Как и EMF, SDO имеет дело с представлением данных в памяти. По существу, реализация SDO представляет собой тонкий уровень («фасад») над EMF, пакетируется и поставляется как часть EMF-проекта.

### 1.5. Технология ADO

Технология ADO (ActiveX Data Objects) разрабатывается компанией Microsoft, первая версия выпущена в 1996 году. Текущая версия ориентирована на платформу .NET [2] и обеспечивает унифицированный доступ к данным различных уровней платформы.

ADO.NET и SDO предназначены для решения одинаковых задач — поддержка XML и приложений с многоуровневой архитектурой. Главным различием между этими двумя технологиями, кроме технических, является то, что ADO.NET предназначена для платформы Microsoft .NET и является проприетарной[4] технологией, в то время как SDO предназначена для платформы Java и стандартизирована через JCP-сообщество.

[4] *Проприета́рное программное обеспечение* (англ. proprietary software) — программное обеспечение, являющееся частной собственностью авторов или правообладателей. Правообладатель сохраняет за собой монополию на его использование, копирование и модификацию.

### 1.6. Шаблон «Serializer»

Шаблон «Serializer» [9] предназначается для обеспечения эффективного сохранения/восстановления (сериализации/десериализации) объектов или наборов переменных в различные структуры данных, например, плоские файлы, таблицы реляционных баз данных, сетевые транспортные буферы и т. д. Эти структуры, в свою очередь, используются для передачи объектов по сети, а также для их сохранения после окончания программы с последующим восстановлением (т. е. реализация свойства persistence). Также шаблон «Serializer» можно применять для реализации функций копирования сложных объектов. Процесс сериализации/десериализации объектов основывается на записи/чтении значений их атрибутов (возможно, с дополнительной информацией в виде имени атрибута и др.) в соответствующие структуры хранения посредством специальных протоколов Reader/Writer. Эти протоколы представляют собой иерархию классов для взаимодействия с различными внешними форматами представления. Прикладное приложение, нуждающееся в сериализации, реализует интерфейс Serializable, который состоит из двух методов — WriteTo(Writer) и ReadFrom(Reader), — предназначенных для взаимодействия с объектами классов Reader/Writer. Существует множество реализаций шаблона «Serializer», например библиотека MFC [10].

Архитектура шаблонов «Serializer» и DAMP имеет общие черты: подобием интерфейса Serializable в DAMP является класс SessionData, а классам Reader/Writer соответствуют DAMP-посредники (mediators), также предназначенные для инкапсуляции доступа к внешним данным. Однако, в отличие от шаблона «Serializer», DAMP не предназначена для сериализации/десериализации объектов. DAMP обеспечивает наложение объектного представления на данные различных типов, выступая в роли «моста» для внешних, независимо созданных данных произвольного формата, представляя их в виде структур и объектов языка программирования в контексте прикладного приложения, использующего эти данные. При сериализации структура данных, в которую происходит сохранение объекта, жестко связана со структурой самого объекта. Это обеспечивает взаимно однозначное соответствие между атрибутами объекта и их сериализованным представлением, что делает возможным последующий процесс десериализации. Однако DAMP взаимодействует с внешними данными, имеющими независимую структуру.

Более того, DAMP в большей степени ориентирована на операции чтения. Запись также поддерживается, но она не рассматривается как сериализация данного объекта, т. е. после операции записи не предполагается возможность восстановления исходного объекта.

### 1.7. Шаблон «Фасад»

Шаблон «Фасад» (Facade) описан в работе [1] и предназначается для предоставления одного унифицированного интерфейса вместо набора разных интерфейсов некоторой подсистемы или набора подсистем. Фасад определяет интерфейс более высокого уровня, который упрощает взаимодействие с данной подсистемой/наборами подсистем.

Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования — свести к минимуму зависимость и унифицировать механизмы взаимодействия подсистем. Использование шаблона «Фасад» позволяет предоставить единый упрощенный интерфейс к более сложным системным средствам.

### 1.8. Шаблон «Посредник»

Шаблон «Посредник» (Mediator) описан в работе [1] и предназначается для инкапсуляции взаимодействия множества объектов друг с другом. Посредник обеспечивает слабую связанность взаимодействующих сторон, избавляя их от необходимости явно ссылаться друг на друга и позволяя тем самым их менять независимо друг от друга.

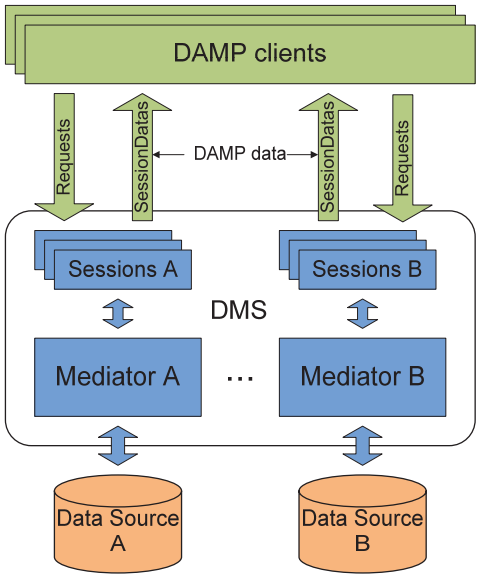
Одной из причин для использования шаблона «Посредник» на практике является наличие объектов в системе, связи между которыми сложны и в то же время четко определены, однако результирующая картина взаимодействия плохо структурирована и трудна для понимания.

## 2. Описание DAMP

Архитектура системы DAMP состоит из следующих основных компонентов (рисунок):

- DAMP-клиенты;
- промежуточные службы данных (Data Mediators Services, DMS);

- объекты данных, DAMP-данные (Data Objects, DAMP data);
- источники данных (Data Sources).




Архитектура системы DAMP.

DAMP-клиенты используют систему DAMP для работы с данными. Взаимодействие с источниками данных осуществляется через промежуточные службы данных (DMS) посредством унифицированного программного интерфейса. Данные могут храниться в любом формате, и только промежуточные службы непосредственно работают с ними.

В свою очередь, DMS представляют собой совокупность двух компонент:

- сессии (Sessions),
- посредники (Mediators).

Сессии являются «фасадами» к подсистемам взаимодействия с данными. В них определяется высокоуровневый универсальный интерфейс для работы с различными внешними источниками данных. Это избавляет DAMP-клиентов от необходимости использовать низкоуровневые интерфейсы подсистем взаимодействия с источниками, которые часто оказываются избыточными по функци-

ональности и сильно затрудняют процесс разработки и сопровождения. Соответствующие специфические программные интерфейсы для каждого конкретного источника инкапсулированы в посредниках. DAMP-клиенты взаимодействуют только с сессиями.

Изначальный формат данных, как правило, малопригоден для непосредственного использования, поэтому в распоряжение клиентского приложения поступают специализированные DAMP-данные, которые бывают двух видов:

- «сырые» DAMP-данные (абстрактный DAMP-класс RawSessionData);
- DAMP-данные, определенные клиентом (абстрактный DAMP-класс SessionData).

С помощью класса SessionData формируется абстрактный уровень представления данных. Важной характерной особенностью класса SessionData является тот факт, что структуру каждого конкретного наследника этого общего типа определяет сам разработчик. Таким образом, разработчик больше обращает внимание на то, в каком виде ему требуются данные из источника, а не на реализацию доступа к ним. Далее, при помощи соответствующей сессии происходит преобразование «сырых» данных из источника к определенной структуре.

В то же время бывают ситуации, когда введение полноценного нового типа в рамках DAMP-приложения по тем или иным причинам нецелесообразно (правда, мы предполагаем, что такие ситуации должны встречаться редко). В этом случае предусмотрено предопределенное представление по умолчанию для данных из источника — реализация абстрактного класса RawSessionData для конкретного типа данных. Как правило, это представление уже в некоторой степени структурировано и определяет базовый интерфейс для взаимодействия с ним, избавляя клиента от низкоуровневых механизмов. То есть для каждого специфического источника данных в рамках DAMP представлена соответствующая реализация.

Библиотека классов системы DAMP предоставляет в распоряжение разработчика полностью реализованный уровень DMS для всех заявленных источников данных и соответствующие RawSessionData. Таким образом, разработчик участвует в разработке только своих собственных SessionData.

Ниже рассмотрим каждый из компонентов и их составляющие более подробно.

### 2.1. Сессии

Опишем «верхний» слой DMS-уровня — сессии. Пусть имеется группа источников данных, с которыми предстоит работать целевому приложению. Как правило, доступ к каждому из них осуществляется посредством платформенно-зависимых библиотек, обеспечивающих формирование сложных низкоуровневых запросов. При использовании DAMP эти библиотеки инкапсулируются в посредниках. В свою очередь, сессии предоставляют единый упрощенный интерфейс к посредникам, который включает только базовые операции доступа к данным: открытие/закрытие, запись/чтение и т. п. Такое ограничение обеспечивает единообразие интерфейса и для большинства реальных задач этих операций оказывается достаточно. Сессии реализуются на основе базового абстрактного DAMP-класса `CSession`, определение которого представлено ниже:

```
[breakindent=134pt] class CSession abstract { public:
   virtual bool   Open   () = 0;
   virtual bool   Close  () = 0;

   virtual CRawSessionData * GetDefaultData() = 0;
   virtual void * GetData() = 0;
   virtual bool   GetData(CSessionData & aDataObj, CKey &
                          aKey) = 0;
   virtual bool   AddData(CSessionData & aDataObj, CKey &
                          aKey) { ... }
   ...
};
```

Конкретные сессии для каждого типа источника данных описываются как наследники этого общего класса, например `CFileSession`, `CHTTPSession`, `CRSSSession` и т. п., которые также входят в библиотеку DAMP. Основные методы класса `CSession` объявлены чисто виртуальными, чтобы гарантировать соответствие конкретных сессий общей модели. Таковым не является метод `AddData()`, потому что, как уже отмечалось ранее, DAMP больше ориентирована на операции чтения данных из разнообразных источников, и реализация этого метода может отсутствовать в конкретном наследнике. Параметрами методов `GetData()` и `AddData()` являются объекты типа `CSessionData` и `CKey` (DAMP-данные), являющиеся парами «ключ — значение». Также предусмотрен метод `GetData()` с пустым списком объектов, передаваемых в качестве

параметров, который взаимодействует непосредственно с RawSessionData, предоставляя «сырые» данные. Использование этого метода позволяет миновать одну ступень, как если бы сначала был вызван метод `GetDefaultData()`, а потом соответствующий метод полученных RawSessionData для доступа к содержимому. Также, например, метод `Open()` может быть параметризован специальным флагом, необходимым для указания режима открытия источника на запись или на чтение.

### 2.2. Посредники

Нижний слой DMS-уровня системы DAMP представлен посредниками, которые используют идею шаблона «Посредник», инкапсулируя в один объект все взаимодействие с определенным источником данных. Никакие другие компоненты или их составляющие системы DAMP не имеют прямого доступа к данным этого источника. Это делает систему более гибкой и устойчивой к изменениям, а также закладывает основу для кроссплатформенности.

Посредники реализуются в виде отдельных сервисных процессов и могут обслуживать много сессий данного типа. Все данные, полученные во время работы с источником (в том числе сообщения об ошибках и т. п.), хранятся в сессиях, и, вообще говоря, посреднику нет необходимости хранить об этом какую-либо дополнительную информацию, кроме списка ссылок активных сессий. В то же время внутри себя посредник может реализовывать специфичные механизмы для поддержки кэширования, оптимизации и т. д. Таким образом, сессии лишь необходимо установить связь с соответствующим посредником и все дальнейшее взаимодействие с источником данных осуществлять через него. Подключение к источнику данных происходит через посредника во время открытия сессии при вызове метода `CConcreteSession::Open()`.

Реализация конкретных посредников может сильно различаться в зависимости от прикладной программы, в рамках которой предполагается их использовать. Например, может потребоваться обеспечить выполнение каждого посредника в отдельном потоке. Общую модель поведения посредников удобно задать в базовом классе `CDataMediator`. Также бывает удобным введение «менеджера» объектов — `CDataMediationManager` — для более гибкого управления объектами типа `CDataMediator`. Такой «менеджер» может представлять собой реестр функционирующих в системе посредни-

ков, отвечая за их корректное создание и удаление в случае сложной структуры и модели поведения, обеспечивая также механизмами реагирования на сбои и т. п. В результате такого разделения на независимые модули вся система в целом становится более гибкой, устойчивой к изменениям и удобной в сопровождении.

### 2.3. Класс SessionData

Данные, полученные из источника, как правило, представлены в «сыром» виде (строка символов, блок байтов и т. п.), в то время как часто удобнее работать с данными, если они уже структурированы для конкретного использования. Необходимая структура определяется разработчиком прикладного ПО, использующим DAMP-подход, в зависимости от решаемой задачи. Для этого система DAMP предоставляет абстрактный базовый тип `CSessionData` — платформенно-независимый уровень представления данных:

```
class CSessionData abstract { public:
  virtual void FillContent (CRawSessionData &
      aRawDataObj, CKey & aKey) = 0;
  virtual void WriteContent(CRawSessionData &
      aRawDataObj, CKey & aKey) {}
};
```

На уровне SessionData формируется объектное представление данных, полученных из источника. Инициализация подобных объектов выполняется на уровне сессии. Поскольку SessionData и сессии распределены по разным уровням архитектуры системы DAMP, необходим способ взаимодействия между ними. В связи с этим для всех наследников класса `CSessionData` является необходимым наличие специального метода, в котором описано преобразование «сырых» данных к требуемому виду — `FillContent()`, который вызывается следующим образом:

```
bool CConcreteSession::GetData(CSessionData & aDataObj,
  CKey & aKey) {
  ...
  aDataObj.FillContent(/*...*/, aKey);
  ...
}
```

Метод `WriteContent()` является обратным к `FillContent()` и по аналогии с `CSession::AddData()` является необязательным.

### 2.4. Класс RawSessionData

Для ситуаций, когда прикладной программе не требуется преобразование к объектам типа SessionData (для работы достаточно «сырого» представления) и по-прежнему остается потребность в простом и эффективном доступе к данным, DAMP предусматривает представление по умолчанию — класс RawSessionData. С одной стороны, это сравнительно простое представление «сырых» данных, с другой — способ учесть всю зависящую от конкретных данных специфику (дескрипторы (handles), временные буферы, указатели и т. п.), доступ к которой должен быть так или иначе обеспечен, иначе модель будет неполна. Еще одно назначение класса RawSessionData заключается в том, чтобы предоставить разработчику удобный интерфейс для работы с данными в «сыром» виде, который можно было бы использовать в методе `FillContent()`:

```
bool CConcreteSession::GetData(CSessionData & aDataObj,
   CKey & aKey) {
   ...
   aDataObj.FillContent(*iRawData, aKey);
   ...
}
```

Объекты класса `CRawSessionData` создаются во время открытия сессии и доступны разработчику через метод `CConcreteSession::GetDefaultData()`.

### 2.5. Ключи

Рассмотрим более детально концептуальную модель представления данных в системе DAMP как множество пар «ключ—значение». Принципиальным элементом этой абстрактной модели являются *ключи* (*keys*). Ключи необходимы для того, чтобы непосредственно в процессе структуризации «сырых» данных указать методу `FillContent()`, какие данные требуются. Наравне с SessionData ключи определяются разработчиком прикладного ПО как наследники базового абстрактного класса `CKey`:

```
class CKey abstract {};
```

Конкретное представление и структура ключа определяются типом данных: так, например, для HTML/XML это могут быть регу-

лярные выражения, для файла — номер строки или байта, для таблицы базы данных — составной ключ, включающий SQL-запросы и дополнительную необходимую информацию. Причем инициализацию ключей можно осуществлять как заранее определенными значениями (в том случае, если заведомо известно, какие именно данные потребуются приложению), так и (что более существенно) различными параметрами, которые являются внешними по отношению к сессии. Поясним последнее подробней.

В рассматриваемом ниже сервере мобильных приложений ключ формируется из параметров, поступивших с мобильного устройства. Например, клиент, работая с приложением, которое отображает курсы валют, выбирает, какие именно валюты его интересуют. Этими значениями и инициализируется ключ, который впоследствии используется в методе `FillContent()` для структуризации именно тех данных, которые были запрошены с мобильного устройства.

Таким образом, ключи обеспечивают ассоциативный доступ к элементам содержимого источника.

## 3. Пример использования DAMP

В качестве примера рассмотрим приложение, которое предоставляет пользователю мобильного устройства курсы запрашиваемых валют. Ежедневные котировки, а также котировки на указанную дату можно получить на сайте ЦБ РФ. Допустим, что ссылка на интернет-базу ЦБ хранится в некотором специальном файле (test.txt). Приложение должно работать по следующему алгоритму:

- получить ссылку на Интернет-ресурс из файла (работает файловая сессия),
- получить котировку валюты по этой ссылке (работает HTTP-сессия),
- структурировать полученную информацию (работает метод `FillContent()`, определенный разработчиком для преобразования «сырых» данных к требуемой структуре),
- обработать информацию — представить ее на экране мобильного устройства.

В контексте рассматриваемой темы нас интересуют три первых пункта.

Определим ключ, который содержит перечень требуемых валют, которые могли быть выбраны пользователем при работе с приложением на мобильном устройстве (например, USD, EUR, GBP):

```
class CCurrenciesKey : public CKey { public:
   CCurrenciesKey() : iCurrent(0)
   {}

   string GetFirst   ();
   string GetNext ();
   ...

private:
   size_t         iCurrent;
   vector<string> iCurrencyCodes;
};
```

Подробности реализации данного и последующих классов мы оставляем за рамками нашего примера.

Внутренняя структура класса `CCurrenciesKey` представляет собой вектор из библиотеки STL, в котором содержатся коды валют. Также нам понадобятся методы `GetFirst()` и `GetNext()` для навигации по элементам вектора.

Определим наследника базового класса `CSessionData` для структурированного представления данных, полученных с сайта ЦБ РФ:

```
class CCurrencies : public CSessionData { public:
   CCurrencies() {}

   virtual void FillContent(CKey& aKey, CRawSessionData&
       aRawDataObj);
   ...

private:
   string            iDate;   // e.g. 06.11.2008
   map<string, string> iRates;   // e.g. (USD, 26.9146)
};
```

Экземпляры класса `CCurrencies` будут содержать дату полученных котировок, а сами котировки будут представляться в виде отображений кодов валют на их стоимость.

Являясь наследником CSessionData, CCurrencies обязан иметь определенный метод FillContent(), в котором описано приведение «сырых» данных к новой структуре. При этом значением параметра aKey будет ссылка на объект CCurrenciesKey, а aRawSessionData будет сообщен сессией во время вызова FillContent().

```
 1 void CCurrencies::FillContent(CKey& aKey, CRawSessionData&
     aRawDataObj) {
 2   CRawHttpData& iRawData = static_cast<CRawHttpData&>(
       aRawDataObj);
 3   CCurrenciesKey& iKey = static_cast<CCurrenciesKey&>(aKey);

 5   // Getting raw content obtained from http://www.cbr.ru
 6   size_t dest_size = 0;
 7   StringCbLengthA(iRawData.GetRawData(),
 8                   STRSAFE_MAX_CCH* sizeof(char),
 9                   &dest_size);
10   char* iBuffer = new char[dest_size+1];
11   StringCbCopyNA(iBuffer,
12                   STRSAFE_MAX_CCH* sizeof(char),
13                   iRawData.GetRawData(),
14                   dest_size+1);

16   // Structurization
17   string iCurrencyCode = iKey.GetFirst();
18   while ( (iCurrencyCode = iKey.GetNext()) != "" ) {
19       string iValue = "";

21       // Parsing iBuffer for current currency code
22       // ...

24       iRates[iCurrencyCode] = iValue;
25   }
26 }
```

Приведенная частично реализация метода FillContent() очень упрощена и для использования в реальном приложении требует значительной доработки.

Метод FillContent() получает ссылки на ключ, в котором содержится информация о требуемых данных, и на объект RawSessionData, который представляет собой базовую структуризацию

«сырого» содержимого источника с удобным интерфейсом для осуществления базовых операций. В нашем примере предполагается, что `CRawHttpData` имеет метод `GetRawData()`, который возвращает данные в том виде, в каком они хранились в источнике. В действительности интерфейс предполагается более гибким и универсальным, например, предоставление данных, соответствующих предоставленным регулярным выражениям. В строках 2–3 производится приведение полученных ссылок к ссылкам соответствующих типов. В строках 6–14 мы получаем содержимое источника с помощью RawSessionData и записываем его в буфер (здесь используются функции для безопасной работы со строками из библиотеки StrSafe.h). Наконец, в строках 17–25 происходит разбор буфера (например, при помощи регулярных выражений) и заполнение внутренней структуры `CCurrencies`. Таким образом, мы определили метод `FillContent()`, вызов которого будет инициирован сессией.

Определенная разработчиком приложения пара «ключ—значение» в рамках DAMP используется следующим образом:

```
CCurrencies    iCurrencies; CCurrenciesKey iKey;

// Setting iKey
// ...

CFileSession iFileSession(TEXT("C:\\test\\test.txt"));
if ( iFileSession.Open() ) {
   CHttpSession iInternet((char*)iFileSession.GetData());
   if ( iInternet.Open() ) {
      iInternet.GetData(iCurrencies, iKey);
   }
} ... iInternet.Close();
iFileSession.Close();
```

Метод `CSession::GetData()` без параметров работает непосредственно с RawSessionData, предоставляя «сырые» данные. В этом примере нет необходимости определять какую-либо дополнительную структуру для данных из файла. Нам необходимо только получить ссылку, поэтому «сырого» содержимого достаточно. Полученная ссылка так же, как и в случае файловой сессии, подается в конструктор HTTP-сессии для инициации соединения.

По завершении работы данного кода, в случае отсутствия ошибок, мы будем иметь определенный нами объект `CCurrencies`, который удобен для дальнейшего использования в приложении.

Стоит отметить, что решение подобной проблемы без использования DAMP потребовало бы от разработчиков значительно больших усилий: пришлось бы реализовывать всю структуру запросов по протоколу HTTP, а также доступ к файлу; нужно было бы отдельно обрабатывать содержимое файла; наконец, мы бы аналогичным образом обрабатывали информацию, полученную с сайта ЦБ РФ. Применив DAMP, мы ограничились только последним, больше сконцентрировавшись на структуризации. Бизнес-логика существенным образом упростилась, что особенно важно, если подобная проблема в контексте разрабатываемого приложения не единична.

## 4. Текущее состояние и статус разработки

На текущий момент в системе DAMP реализована поддержка файловых и HTTP- источников данных. Библиотека включает следующие классы:

- `CFileSession:CSession` — файловая сессия;
- `CHttpSession:CSession` — HTTP-сессия;
- `CDataMediationManager` — «менеджер» объектов типа `CDataMediator`;
- `CFileMediator:CDataMediator` — посредник для взаимодействия с файловыми источниками;
- `CHttpMediator:CDataMediator` — посредник для взаимодействия с HTTP-источниками;
- `CRawFileData:CRawSessionData` — файловые RawSessionData, хранящие «сырое» представление файлов во внутреннем буфере и предоставляющие удобный интерфейс для взаимодействия с ним;
- `CRawHttpData:CRawSessionData` — аналогично `CRawFileData`;
- `CTextFile:CSessionData` — построчное представление файлов в терминах DAMP-данных.

В качестве средства программирования используется Microsoft Visual C++ 2008. Соответствующие посредники реализовывались при помощи библиотек WinBase (для работы с файлами) и WinHttp (для работы по протоколу HTTP). В ближайшем будущем планируется реализовать поддержку RSS-источников, баз данных, а также включить поддержку различных Web-служб.

Кроме того, система DAMP используется в качестве основного программного интерфейса системы «Ubiq Mobile» для доступа к внешним данным.

## Заключение

В некоторых задачах может возникнуть потребность в более специфичных функций работы с данными, выходящих за рамки предоставляемого DAMP-сессией унифицированного интерфейса CRUD. Такой доступ может быть предоставлен разработчику через посредников. Однако в таком случае пропадает эффективность процесса получения и структуризации данных, которой обеспечивается сессией. При разработке приложений с применением DAMP прямое использование посредников целесообразно только в исключительных ситуациях, когда остро стоит задача эффективности.

В данной статье система DAMP рассматривалась с точки зрения операций получения данных из внешних источников. Однако описанный подход подразумевает также поддержку и операций записи. Но традиционно запись является более сложной операцией, поэтому возможны дополнительные ограничения при преобразовании структурированного представления данных (SessionData) к изначальному («сырому»). Этот вопрос остается открытым и на данный момент является предметом исследования. В дальнейшем планируется полноценная поддержка режимов записи и модификации данных.

## Список литературы


[1] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2007. 366 с.

[2] *Малик С.* Microsoft ADO.NET 2.0 для профессионалов. М.: Изд. дом «Вильямс», 2006. 560 с.

[3] *Bell M.* Service-Oriented Modeling: Service Analysis, Design, and Architecture. Hoboken, NJ: John Wiley & Sons, Inc., 2008. 384 p.

[4] Enterprise Information Systems (EIS). http://www.compinfo-center.com/tpeis-t.htm

[5] Introduction to Service Data Objects, September, 2004. http://www.ibm.com/developerworks/webservices/library/j-sdo/

[6] Java Architecture for XML Binding (JAXB), March, 2003. http://java.sun.com/developer/technicalArticles/WebServices/jaxb/

[7] Java Data Objects (JDO). http://java.sun.com/jdo/

[8] *Panda D., Rahman R., Lane D.* EJB 3 in Action. Greenwich, CT: Manning Publications Co., 2007. 712 p.

[9] *Riehle D., Siberski W., Baumer D. et al.* Serializer // Martin R.,

Riehle D., Buschmann F. Pattern Languages of Program Design 3. Massachusetts: Addison-Wesley Professional, 1997. P. 293–312.

[10] Serialization in MFC. http://msdn.microsoft.com/en-us/library/6bz744w8.aspx

[11] *Steinberg D., Budinsky F., Paternostro M., Merks E.* EMF: Eclipse Modeling Framework (2nd Edition). Boston, MA: Pearson Education, Inc., 2008. 744 p.