Data Access Management Pattern

Sergey Skudarnov
Saint-Petersburg State University
Peterhof, Russia
sergey.skudarnov@gmail.com

Abstract

The paper describes basic ideas, architectural structure and details of implementation of the pattern for designing uniform interaction model with external data sources and external services – called Data Access Management Pattern – that has been developed within the scope of Ubiq Mobile research project – a universal platform for mobile online services. The pattern presents completely uniform API for all supported types of external data sources. Such unification imposes some functional limitations but there are use cases when the basic set of operations for processing external data – such as Open and Close, Read and Write, Seek, etc. – is generically enough. In those cases wide support of external data sources is more preferable. The principal novelty of our pattern is that it provides client with the ability to work with external data in customized representation, specially tuned for concrete clients' needs.

INDEX TERMS: DATA ACCESS, EXTERNAL DATA SOURCES, DESIGN PATTERN, UBIQ MOBILE PLATFORM

I. INTRODUCTION

Systems that need to work intensively with various types of external data sources (like HTML and XML, external files, databases, and so forth) and external services (TCP/IP, RSS feeds, Web services, and so forth) required a uniform interaction model. We will use a "data source" term for both external data sources and external services when talking about external data in general and differentiate them explicitly if necessary. Such sort of unification problems are used to solve through design patterns. The key requirements to the design pattern that could be applicable for the situations like this one are following:

- Data access unification: The pattern have to provide uniform API for all supported types of data sources;
- Data access management: The pattern have to manage synchronization issues and shared access policies;
- Data abstraction and manipulation: The pattern should provide client with the ability to work with data in abstract object-oriented terms.

The problem of uniform interaction with external data sources arises in many software development projects. The most common way to solve it is development of restricted ad hoc solutions within the scope of specific projects to meet specific needs. On the other hand there are specialized frameworks like JDO (Java Data Objects) [1], SDO (Service Data Objects) [2], ADO (ActiveX Data Objects) [3], and so forth. Such frameworks have rich functionality (even redundant in some cases) but their applicability is limited for different reasons: the frameworks are too "fat" and heavyweight due to their universality and rich functionality,

support only restricted set of data sources, are platform-dependent, etc. But there are many situations where only basic set of operations for access to external data – such as Open and Close, Read and Write, Seek, etc. – is needed. This article is targeted to systems where unification is more important in comparison with rich functionality. Such cases can include, for example, various platforms and frameworks that provide uniform API for access to external data sources. The paper describes Data Access Management Pattern (DAMP) – a pattern that presents one of possible approaches to design the uniform interaction model with heterogeneous data sources.

The DAMP pattern presents limited but self-sufficient functionality for access to external data sources via uniform interfaces for all supported types of data. At the same time DAMP pattern allows the user to customize external data, structuring them in his/her personal way. In such case all standard DAMP operations would work with structured data performing necessary filtering and marshalling actions automatically.

Interfaces with external services are more diverse than interfaces with the data sources, and therefore cannot be formalized within a single pattern. For some types of services providing sort of data access (like RSS) we can use universal DAMP pattern whereas for more complex services it is impossible and the related pattern's components provide their own API.

During the rest of paper we will use "DAMP pattern" term meaning a general design structure, and "DAMP system" term as the pattern's implementation.

The work has been fulfilled within the scope of Ubiq Mobile project [4] – a universal platform for mobile online services – that uses DAMP system as a part of its server software for providing user applications with uniform access to external data sources.

II. DAMP Architecture

The DAMP architecture provides a set of core components (see Figure 1):

- Data Mediator Services
- Data holders (DAMP Data Objects)

DAMP clients are programs and applications running on the server that use DAMP system for access to the data. Instead of using technology-specific or platform-specific APIs and frameworks, DAMP clients use relationally simple DAMP programming model and uniform API provided by Data Mediator Services (DMS). DAMP clients work in terms of DAMP Data Objects and do not need to know additional details regarding data persistence, serialization, etc.

In turn DMS includes the following types of components:

- Mediators
- Sessions
- Descriptors

A. Mediators

All interactions of DAMP clients with data sources are performed through Mediators – service processes that constantly run on the server and work with users' applications in the request-response mode. Mediators provide DAMP clients with access to external data (like HTML and XML, external files, databases, and so forth) as well as to external services

(TCP/IP, RSS feeds, Web services, etc.). Each Mediator works with its own type of data source serving all DAMP clients. Mediators encapsulate all data source-specific interaction routines being the only components inside DAMP system that have direct access to the data sources. Such approach makes the whole DAMP system more flexible in terms of extensibility by adding support for new types of data sources and resistance to possible changes.

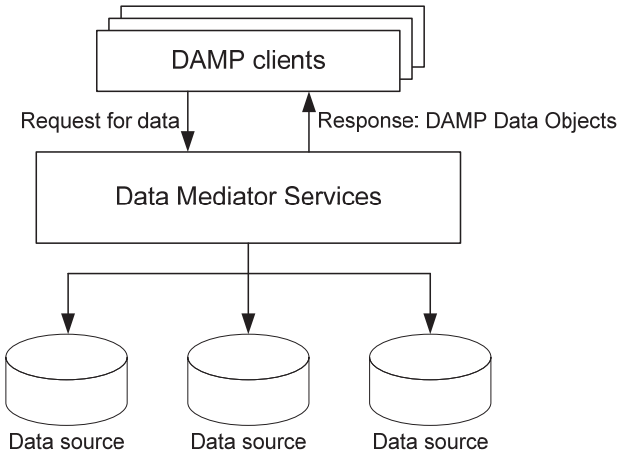

Fig. 1. General architecture

Mediators are low-level components of DAMP system that encapsulate all data source interaction routines. They are obviously data source-dependent and so cannot provide abstract and uniform data source interaction model. DAMP pattern contains interface components - Sessions and Descriptors – that provide required unification.

B. Sessions

DAMP clients don't have direct access to Mediators' methods and data. Instead, they are working with Mediators via objects of special classes – Sessions and Descriptors – provided by platform API.

Sessions act as connection providers. When DAMP client needs access to external data source, it uses an appropriate Session (corresponding to the type of data, one Session works for all data sources of that type) to establish a connection with the data source. The Sessions provide basic API for connection and disconnection operations that is completely uniform and similar to all supported data sources.

For access to the specific data source of a given type, the DAMP client establishes connection with the data source through a Session. After establishing connection, the Session returns to DAMP client a special object called Descriptor that represents particular data source.

C. Descriptors

Descriptors are special DAMP objects that "represent" external data sources for applications in the same manner like handles represent files in Win32 API. When Session provides access to a data source, it assigns a Descriptor for the access and the Descriptor is

being associated with the source until either the Session terminates or the Descriptor is closed using the Session's API. Like Sessions, Descriptors provide uniform API for all supported types of data sources.

Descriptors' creation and destruction operations are managed by Sessions. So for usability purposes it is convenient to use Factory Method pattern [5] targeted to such sort of relationship. Descriptors are considered as products whereas Sessions act as Descriptors' creators. The Session class is an abstract class and does not provide an implementation for the factory method it declares. Each subclass defines an implementation for the factory method that creates the appropriate Descriptor. So we have a parallel class hierarchy that represents the relationship between Descriptors and Sessions (see Fig. 2).

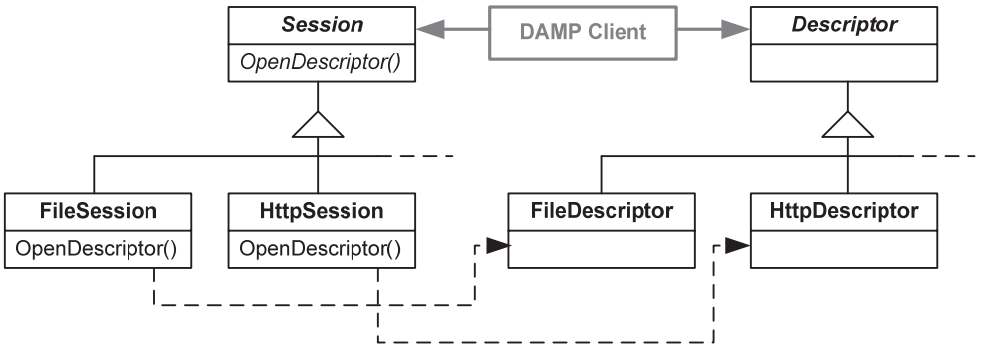

Fig. 2. Descriptors and Sessions class hierarchy

In the DAMP system Descriptors are used not only to identify the data sources. All data retrieved from the data sources using the Mediators are temporarily stored in special objects – instance of RawSessionData (RSD) class. Descriptors use RSD objects as their internal buffers and manage their creation, destruction and life-time issues.

D. RawSessionData

RawSessionData is one of the DAMP Data Objects type. Data retrieved from the external data source by Mediators is represented in its native format (we call such data as raw data). The raw data is stored in the RSD objects that are managed by Descriptors. RSD objects act as intelligent buffers with a basic set of operations to work with its content. In the DAMP system for each supported type of data sources a concrete RSD class derived from the base class RawSessionData is created. The internal structure of each derived RSD is specific to the concrete data source and can implement its specific storing/caching/interacting behaviors and policies. The interface of each derived RSD is also dependent on the concrete data source.

RSD objects are internal components in the DAMP system and in most cases DAMP clients don't need direct access to them. Instead, they work with data through another type of DAMP Data Objects – SessionData.

E. SessionData

One of the DAMP pattern's key points is to provide DAMP client with the ability to work with external data in customized representation specially tuned for concrete clients' needs.

Such customization can be performed using special objects – SessionData. There are two possible SessionData's use cases.

The DAMP system assumes set of predefined SessionData objects for most commonly used types of data sources like files, web pages (accessed via HTTP(S) protocol), and so forth. Session API provides the access to those SessionData objects. So in standard cases DAMP clients may simply choose corresponding Session to get required SessionData object from the set of predefined ones.

Another use case is to define your own SessionData if no predefined ones are suitable for your needs. Client-defined SessionData must meet certain rules to be compatible with DMS (usually it is implementing via the pure virtual inheritance). That rules include the demand for implementation of special method – `FillContent()` – that defines conversion process from raw data to SessionData representation. Without defining `FillContent()` it is impossible to make customization because DMS don't know how to make data conversion. All conversion routines are encapsulated in `FillContent()` method, so DMS should simply invoke it to perform the conversion. For predefined SessionData objects `FillContent()` methods are also predefined, so one doesn't need to take care of it. Once the new SessionData is defined, DAMP client uses Descriptor API to invoke `FillContent()`.

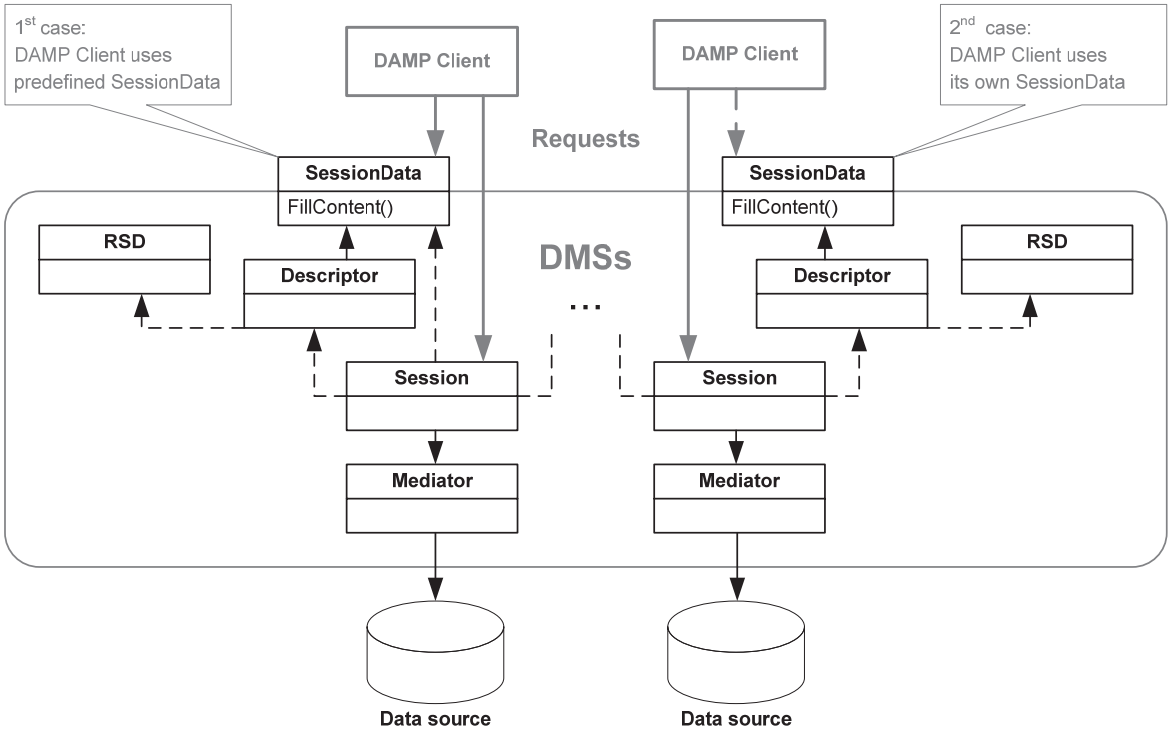

Fig. 3. General interacting structure

III. SAMPLE CODE

In this section we describe the two possible ways of DAMP system usage. We are focusing only on client-side code, leaving all internal DMS-level implementation behind the scenes. As the programming language, C++ has been chosen.

In the first example we use predefined SessionData object for access to an ordinary text file. `Session` class declares operations for Descriptors' construction and destruction. Subclasses implement these operations for specific kinds of Descriptors. Also each subclass defines operations for predefined SessionData objects usage. In our example we have `OpenTextFile()` method that returns an object of `TextFile` type – a subclass of abstract `SessionData` class.

```
enum OpenMode { ERead, EWrite }; //flags to specify data source's open
mode

class FileSession : public Session {
public:
    // ...
    virtual FileDescriptor* OpenFile(string fileName,
                                     OpenMode openMode = ERead);
    virtual bool CloseFile(Descriptor* aDescriptor);
    TextFile* OpenTextFile(string fileName, OpenMode openMode = ERead);
    // ...
};
```

`TextFile` object defines a set of operations that are typical for text files processing. Here is an example of possible interface.

```
class TextFile : public SessionData {
public:
    // ...
    string GetFirstLine();
    string GetNextLine();
    string operator[](size_t lineNo);
    bool Seek(size_t lineNo) const;
    size_t GetCurrentLineNo() const;
    size_t Size() const;
    // ...
};
```

The client code will be following:

```
FileSession fileSession;
TextFile* textFile = fileSession.OpenTextFile("example.txt", ERead);
// textFile usage...
delete textFile;
```

Notice that for Descriptors we have explicit Close operation. For SessionData objects there is no such operation, so the client is responsible for correct resource release.

In addition to the predefined SessionData objects, we can use client-defined SessionData objects to convert raw data into customized representation. As an example, let's consider HTML page source. DAMP client should define a `SessionData`'s subclass for its customization – let's name it `MyHtml`. `SessionData` is an abstract class that declares method `FillContent(RawSessionData&)` for performing converting operations. Each subclass of `SessionData` must provide an implementation for this method where conversion procedure from raw data format into client's customized data representation is defined.

`FillContent()` takes a reference to `RawSessionData` object as a parameter that can be used inside `FillContent()` for conversion implementation.

```
class MyHtml : public SessionData {
public:
    virtual void FillContent(RawSessionData&);
    // other methods, fields, and so forth...
};

void MyHtml::FillContent(RawSessionData& rawSessionData) {
    RawHtmlData& rawData = static_cast<RawHtmlData&>(rawSessionData);
    // further implementation...
}
```

Once `MyHtml` is defined we use corresponding Descriptor to execute `FillContent()` method.

```
class HttpDescriptor : public Descriptor {
public:
    bool GetData(SessionData&);
    // ...

private:
    RawHtmlData* rawData; // class RawHtmlData : public RawSessionData
    // ...
};

bool HttpDescriptor::GetData(SessionData& sessionData) {
    sessionData.FillContent(*rawData);
    return true;
}
```

So the client code will be following:

```
MyHtml myHtml;
// ...
HttpSession httpSession;
HttpDescriptor* descr = httpSession.OpenHttp("http://...", ERead);
Descr->GetData(myHtml);
httpSession.CloseHttp(descr);
// myHtml usage...
```

IV. Conclusion

The DAMP system has been developed in the Department of Applied Cybernetics of St. Petersburg State University as a research project. Currently there is a version of the system that supports such external data sources as files and HTML pages (accessed via the HTTP protocol). The DAMP system is integrated into the server part of the Ubiq Mobile platform. University students developed several test applications that we use for demonstration and debugging purposes. Currently only read-only access to the data sources is supported.

The plans for the nearest future include adding support for other external data sources like RSS feeds, data bases, and so forth. Also we are still working on writing operations. Writing is actually more complex operation than reading and it can imply some nontrivial pitfalls. But

in the future versions of the DAMP system we are planning to add completely support for both reading and writing operations.

REFERENCES

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

[2] Simplify and unify data with a Service Data Objects architecture, September, 2005, http://www.ibm.com/developerworks/webservices/library/ws-sdoarch/

[3] Jason T. Roff, "ADO: ActiveX Data Objects", *O'Reilly Media*, 2001.

[4] Onossovski Valentin, Terekhov Andrey, "Ubiq Mobile – a New Universal Platform for Mobile Online Services", Proceedings of 6th Seminar of Finish-Russian University Cooperation in Telecommunications (FRUCT) Program, Helsinki, Finland, 3-6 November 2009, pp. 96-105.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", *Addison-Wesley Professional*, pp. 107–117, 1994.