

Business Logic on the Client-Side

Design Patterns on the Implementation of Business Logic in Client Applications

Tim Wellhausen

kontakt@tim-wellhausen.de
<http://www.tim-wellhausen.de>

Conference draft for EuroPLoP 2006

A note to the participants of the EuroPLoP writers' workshop:

I'd like to encourage a discussion about the names of the patterns. Are they appropriate for use in daily work? Can you come up with better names?

I'd also like to know your opinion about the current implementation sections. Are they too closely bound to the running example? Would you prefer implementation sections that are based on participants and collaborations?

Are any patterns missing?

Abstract: It is a general rule of thumb that business logic should be implemented in a middle-tier component and be separated from client applications. Although this rule has demonstrated its validity, it restricts the options for designing information systems. At times, there may be good reasons to deploy some types of business logic locally in client applications.

This paper presents a collection of design patterns that address the forces of implementing business logic in client applications in the context of an object-oriented business domain layer.

Introduction

In an information system, the rules and activities of a real-world business domain are represented by *business logic*¹. A corresponding *business domain model* defines the associations and properties of *business data*. The business logic manipulates the data and, at the same time, ensures their consistency and validity.

In principle, business logic should be implemented in a business domain layer to separate the business logic source code from the user interface and technical infrastructure source code. Accordingly, the user interface should only present business data and let users start the execution of business logic.

There are practical reasons, however, not to isolate business logic completely from client applications. To make a client application convenient to use, some information about the business domain has to be incorporated into the client application.

These are some examples:

- *Responsiveness*. User input must be validated before its processing, for example a birth date or a credit card number. To make an application more responsive, some validations may need to be performed locally.
- *Performance*. Some periodically performed computations may require the evaluation of large amounts of data, for example the analysis of stock quotations. A stateless server component would need to fetch the entire data required for the computation upon each invocation. A rich client application may locally store and update the available data and perform the same calculations without additional costs.
- *Presentation*. Business data may need to be prepared for presentation. This includes the textual representation of business entities or the graphical representation of calculations, for example the presentation of a schedule. How business data is presented may vary between client applications and typically is not an essential part of the business logic.

* * *

Although multi-tier information systems can be decomposed in many different ways, the implementation of a LAYERED ARCHITECTURE ([Evans 2004]) has become wide-spread. Such an architecture mandates a separation of concerns that involves at least separating the presentation layer from the business domain and infrastructure layers.

Typically, business domain and infrastructure layers are deployed in application servers whereas the presentation layer is part of a web or rich client application. Each layer has a particular responsibility:

- The presentation layer handles user interactions and calls the business domain layer on user requests.
- The business domain layer contains the business logic and exposes publicly available services to make it available to client applications.

¹ In the literature, the terms *business logic* and *domain logic* are often used synonymously. In this paper, only the term *business logic* is used.

- The infrastructure layer retrieves and stores data and provides access to external systems.

How a business domain layer and an object-oriented business domain model are built varies tremendously. This is dependent not only on the scope of the project but also on the actual requirements and the available budget and time. Some typical variations are provided below:

A simple domain model may only consist of business objects with primitive values. In this case, for example, the columns of a database table directly correspond to the fields of a business object. However, relations between database tables are not mapped to associations between business objects.

A more sophisticated domain model may employ object associations and polymorphism. In this case, the information system may support loading and saving graphs of business objects, including object associations, and support the usage of polymorphism to create business object class hierarchies. A client application may then retrieve an expanded graph of business objects in one call.

Whether business objects actually contain any business logic or not may also vary. If business objects do not contain business logic, server-side business logic is often implemented in the form of use cases that load, modify, and store business objects.

Whether a client application uses the same business objects as are used on the server-side may vary as well. If business objects are not sent to client applications, typically, DATA TRANSFER OBJECTS ([Marinescu 2002]) are exchanged between the server-side and the client-side.

The source code of business objects may be generated by a code generation tool. Particularly if a project team applies a model-based or model-driven development approach, the source code for business objects is seldom hand-crafted.

* * *

Within the context of an object-oriented business domain model on the server-side, the following problem often needs to be addressed:

How do you implement business logic that is primarily needed on the client-side?

Common Problem

Although business logic should reside in a business domain layer and be isolated from other layers of a system, some types of business logic may be needed locally by client applications. The following forces constrain the implementation of such business logic:

Common Forces

Logic that is needed on the client-side often is small and easy to program. It is therefore frequently and redundantly implemented in many different places. In particular, the business logic source code in a client application tends to be interwoven with the user interface source code for event handling, causing a maintenance nightmare. If the GUI changes, the business logic may require adjusting as well. If the business logic is modified, it may become complicated to locate all spots in the GUI that need be addressed.

If, on the other hand, a strict standard is enforced to keep business logic completely on the server-side, other problems may arise. Some logic may be relevant only to client applications. If such a logic requires frequent modifications during the devel-

opment of the client application, the process of development for client and server components becomes closely coupled.

Furthermore, if a client application must call some business logic frequently, fine-grained remote method calls for many user operations would probably cause unacceptable delays in the user interaction and result in increased network overhead.

Deploying business logic in a client application may evoke security issues if modifications to business data are not checked on the server-side as well. Malicious users may modify the client applications to let them ignore security checks.

In a multi-user environment, the client application may not be able to enforce the integrity of business data. Only a centralized server component is able to handle cases of synchronous user actions.

* * *

The following patterns resolve these common forces. Note that the patterns do not complement each other; they are rather alternatives for solving a common problem, each of which may have a broader context and additional forces.

**Thumb-
nails**

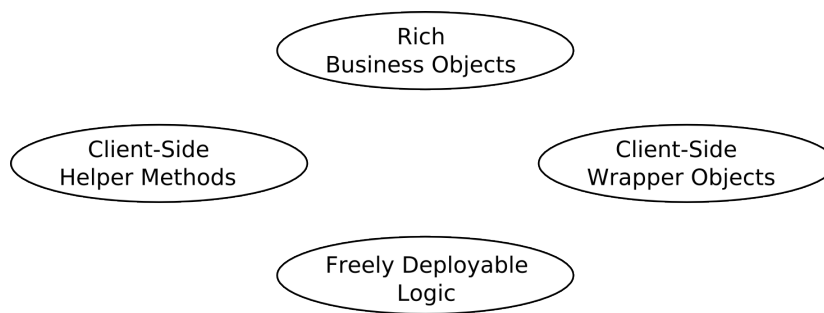


Fig. 1: The Patterns

You may implement client-side business logic in RICH BUSINESS OBJECTS as it is the object-oriented way of coupling data and associated behavior.

If the client application only receives pure data transfer objects and a simple solution is essential, you may implement business logic in CLIENT-SIDE HELPER METHODS.

If, likewise on the client-side, object-oriented concepts add significant value over procedural code despite exchanging pure data transfer objects, you should consider the implementation of business logic in CLIENT-SIDE WRAPPER OBJECTS.

Business logic may not be intended to be implemented statically in a client application and, at the same time, a decision regarding where to deploy the logic may be desired at a later time. Then you should consider FREELY DEPLOYABLE LOGIC, which may be deployed on both client-side and server-side.

Running Example

A running example illustrates the patterns. Imagine a music player that is locally deployed on a client machine and connected to a server from which it fetches play lists and retrieves information about albums. The server-side business domain model for this example is depicted in Fig. 2. Note that although the running example is illustrated in Java, the patterns are not restricted to Java.

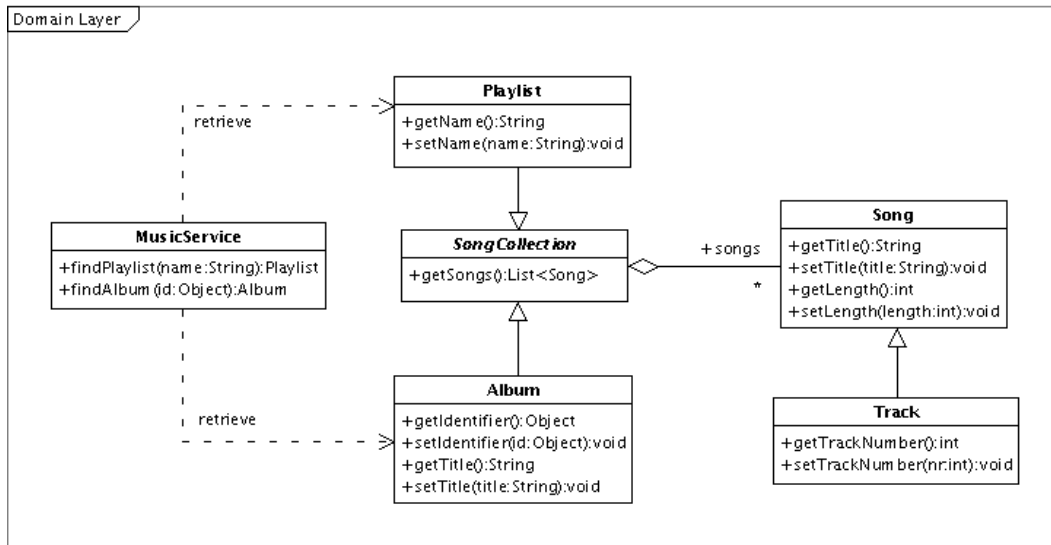


Fig. 2: Server-side domain model of a music player

A `MusicService` provides access to the business data by letting a client application retrieve business objects by their keys. The client application may retrieve instances of `Album` and instances of `Playlist`. Both `Album` and `Playlist` have a common super class, `SongCollection`, whose responsibility basically is to hold references to songs.

A `Song` object has a title and a length in seconds. If a song is part of an album, the `Album` instance keeps references to instances of `Track` that additionally keeps the track number. An `Album` has a unique identifier and a title, a `Playlist` a name.

Moreover, suppose the following requirements applied to the client application:

- When displayed, the name of an album and the title of a play list should always be extended by the number of songs and their overall length.
- The name of a song should always be shown in conjunction with the length of the song.
- The list of tracks that an album refers to must be sorted according to the track enumeration.

Each requirement involves some data processing that is based on the business objects transmitted to the client application.

Rich Business Objects

Implement client-side business logic in those business objects that are sent from the server component to the client application.

As a constrictor to the common context, client applications receive the same business objects as are used on the server-side.

Additional Context

* * *

The object-oriented paradigm has proven to be very powerful. One of its core insights is to KEEP RELATED DATA AND BEHAVIOR IN ONE PLACE ([Riel 1996]). If data and behavior are separated, business logic may become splattered over many places, which makes it difficult to keep track of it.

Additional Forces

Therefore:

Keep business logic in business objects. Enrich each business object with the business logic needed by the client application.

Solution

Give each business object a clear and narrow responsibility. Each operation should be implemented in that object that holds or may gather the relevant data. Use polymorphism to implement behavior that differs in sub classes.

To apply this pattern, decompose client-side business logic into the appropriate business objects. Fig. 3 shows how the example classes are extended by client-side business logic (new methods are marked in blue color).

Implementation

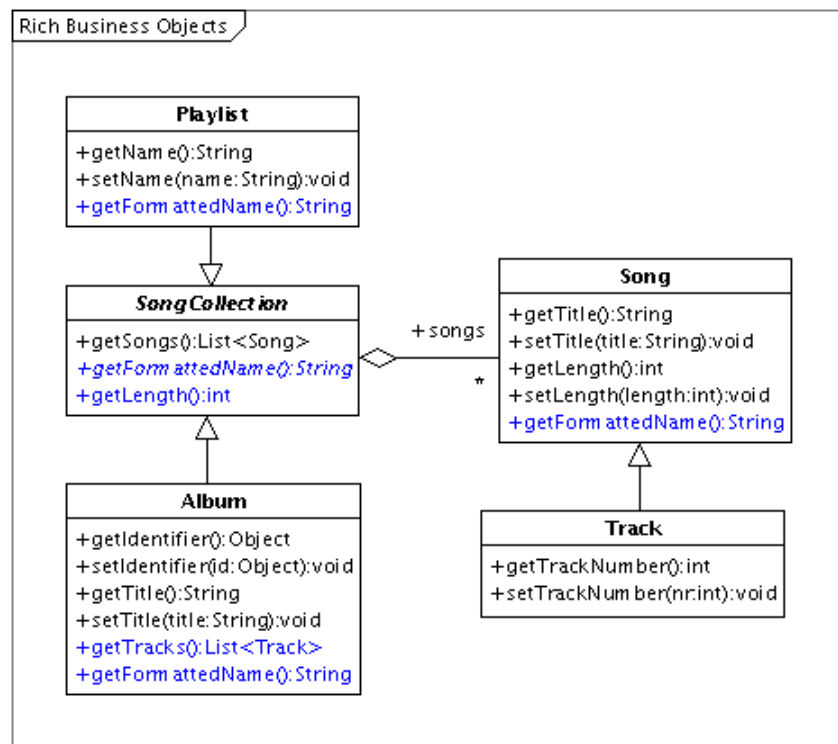


Fig. 3: Example resolved using Rich Business Objects

SongCollection got two operations: getLength and getFormattedName. The method getFormattedName has been made abstract because the formatted names

of play lists and albums differ. The accessor method `getSongs` was overridden in `Album` to ensure that the list of tracks is always sorted.

* * *

Implementing business logic in business objects keeps the logic and the data together, even on the client-side. If this pattern is applied strictly, business logic is kept out of GUI code, therefore easier to locate and maintain. The complete client-side business logic is explicitly defined by business object interfaces.

**Con-
sequences**

This approach has severe drawbacks, though. Implementing client-side business logic in business objects that are managed by a server component violates the separation of concerns: a client application may interfere with the responsibilities of the server component, possibly invoking security and integrity issues.

Business objects that are transferred to a client may provide high-level operations that expose server-side business logic to the client application. Such operations may not be called there directly, e.g. outside a container.

Whenever changes to client-side requirements lead to business object code changes, the server component must be re-deployed. As a consequence thereof, the development process of client-side and server-side must be closely coordinated.

There is a risk of coupling the server component with client-side classes from a GUI library if, for example, a list of rich business objects implements a GUI list model interface. However, server components should not know any details of client applications.

A server component may be called from several client applications, each of which may require different kinds of business logic. Sharing the same business objects may multiply development and deployment conflicts.

If a model-based development approach is applied, the source code of business objects is often generated from a high-level abstraction of the business domain model (see, for example, [Völter et al. 2004]). Depending on the product used for code generation, it may be difficult to merge hand-written business logic with generated source code.

The more business logic is enriched, the bigger business objects tend to become. Often, a few dominant classes contain most of the code. These classes sometimes deteriorate into God classes (see [Riel 1996]).

Client-Side Helper Methods

Implement business logic in static helper methods that are deployed locally in client applications.

Some business logic is only needed on the client-side, for example for the presentation of business data.

**Additional
Context**

Pure data transfer objects are exchanged between the client-side and the server-side to reduce interdependencies and simplify deployment.

* * *

Sometimes, a simple solution is the best solution because budget or time restrictions prevent implementing a clean and elegant solution to handle client-side business logic.

**Additional
Forces**

Therefore:

Create dedicated helper classes that are available only in client applications and implement business logic in static methods that operate on given business objects.

Solution

For each business object, create a helper class to encapsulate business logic that operates on the business object. You may omit helper classes for those business objects for which no client-side business logic exists.

If the business domain model incorporates polymorphism, you may create a helper class for the base class, for some classes of the hierarchy, or for each class. There is a trade-off between a proliferation of helper classes and the cost of introducing conditional statements.

To make the connection between business objects and helper classes as explicit as possible, the helper classes should mirror the structure of the business domain model, for example by applying a similar package naming schema.

**Imple-
mentation**

Static helper methods should be stateless by default. Only if the results of some computations are frequently requested and the data these computations rely on seldom change, you may need to implement a local, static caching mechanism.

Fig. 4 contains a possible implementation for the running example.

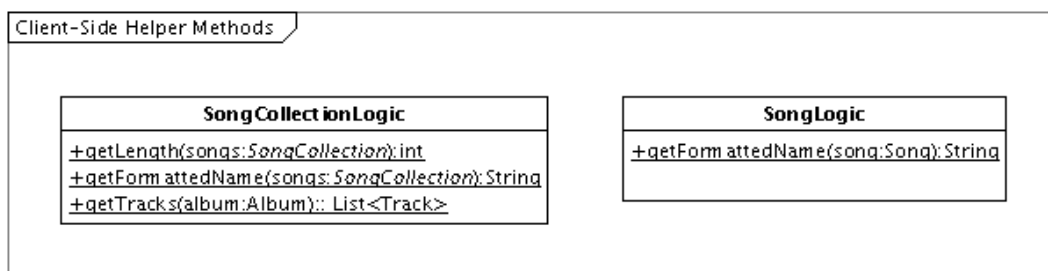


Fig. 4: Example resolved using Client-Side Helper Methods

The business logic for SongCollection and its sub classes Playlist and Album is encapsulated in SongCollectionLogic. The implementation of getLength operates on a given SongCollection whereas in getFormattedName, a conditional distinction between Playlist and Album is necessary. There is no need for a

class `TrackLogic` because there is no business logic particularly for `Track` instances.

* * *

By applying such a procedural approach, changes to client-side helper classes only affect the respective client application. Business logic for the client-side is easily added and modified.

**Con-
sequences**

The development process of business logic on the server-side is not coupled to the development process of business logic on the client-side. If several distinct client applications communicate with the same server component, each one may implement its own set of helper methods. Then, a close coupling between each involved system is also avoided.

As a downside, business data is separated from its behavior. Static helper methods are often seen as a symptom of a badly designed object-oriented system. They are easy to create, the advantages of object-oriented programming are lost, business logic code may get duplicated, and the reusability of business logic may be reduced.

Client-Side Wrapper Objects

For each business object, create a wrapper object that contains the business logic.

Some business logic is only needed on the client-side, for example for the presentation of business data.

**Additional
Context**

Pure data transfer objects are exchanged between the client-side and the server-side to reduce interdependencies and simplify deployment.

The business domain model is complex and contains deep class hierarchies and many associations.

* * *

Implementing business logic in CLIENT-SIDE HELPER METHODS may cause business logic to deteriorate into conditional code that is difficult to understand and maintain.

**Additional
Forces**

Therefore:

For each business object, create a corresponding wrapper object that contains the business logic and acts as a proxy to the actual business object.

Solution

Wrapper objects receive references to the wrapped business objects upon creation and then provide access to the wrapped business data. For each association of a wrapped business object, a new association to the wrapping equivalence has to be maintained.

Whenever the client application retrieves business objects from the server component, the system must create appropriate wrappers and return them instead of the actual business objects. Consider using the BUSINESS DELEGATE pattern (Alur et al. 2003) to encapsulate this process.

To implement this pattern, a wrapper must be created for each business object, as shown in Fig. 5. The upper half of the class diagram shows the business objects from Fig. 2 (Playlist has been left out in this example). The lower half contains the corresponding wrappers: one for each business object (marked in blue color).

**Imple-
mentation**

The wrappers form a class hierarchy that corresponds to the hierarchy of business objects. At runtime, each wrapper refers to a wrapped object, and for each association in the business domain model, there is an association between wrappers.

In the example, every SongWrapper refers to a Song and every SongCollectionWrapper refers to both a SongCollection and a list of SongWrapper objects. Besides these references, SongCollectionWrapper contains business logic that applies to SongCollection objects and SongWrapper business logic that applies to Song instances.

For the sub classes of SongCollection and Song, Album and Track, respectively, there are additional wrappers: AlbumWrapper and TrackWrapper. Explicit references to their wrapped objects are not needed as they inherit the references from their base classes. However, both classes provide type-safe getter methods to the wrapped objects (getAlbum(), getTrack()).

The creation of wrapper objects should be delegated to a FACTORY ([Evans 2004]). Both the business delegate objects that retrieve business objects from the server component and the wrappers themselves should call the wrapper factory for object

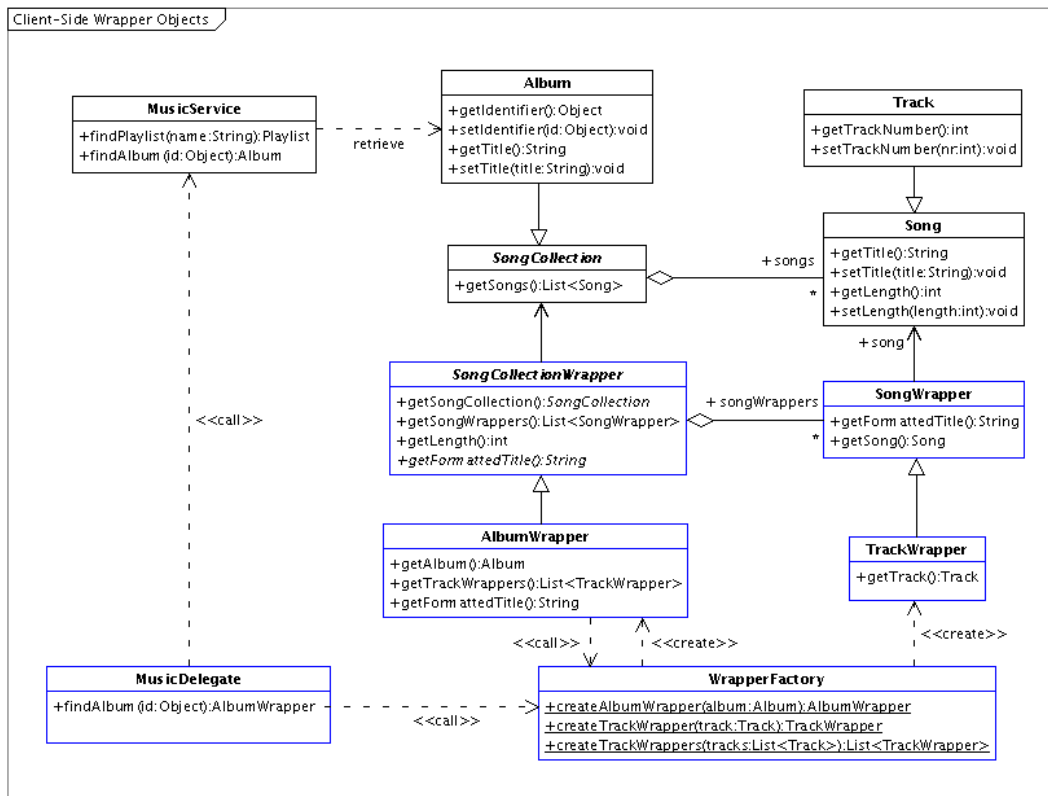


Fig. 5: Example resolved using Client-Side Wrapper Objects

creation. In the example, the business delegate `MusicDelegate` calls the `WrapperFactory` to let it create an instance of `AlbumWrapper` for every `Album` object received:

```
public class MusicDelegate {
    public AlbumWrapper findAlbum(Object id) {
        Album album = getMusicService().findAlbum(id);
        return WrapperFactory.createAlbumWrapper(album);
    }
}
```

The implementation of `AlbumWrapper.getTrackWrappers()` first also delegates to the factory and then fulfills the requirement to return a sorted list of tracks:

```
public class AlbumWrapper {
    public List<TrackWrapper> getTrackWrappers() {
        List<TrackWrapper> trackWrappers =
            WrapperFactory.createTrackWrappers(getTracks());
        Collections.sort(trackWrappers, getTrackComparator());
        return trackWrappers;
    }
}
```

Using a factory makes it possible to switch the creation strategy of wrapper objects. If it is important to limit the maximum memory consumption, the factory may create new wrapper objects upon each request so that these objects may be disposed by the garbage collector soon after the call. If it is important, on the other hand, to increase the operational speed, the factory may keep the created wrapper objects in a cache.

To increase the convenience for client programmers, getter methods may be added to the wrappers to make the properties of the business objects more easily retrievable, as shown in the following example:

```
public class AlbumWrapper {
    public String getTitle() {
        return getAlbum().getTitle();
    }
}
```

Furthermore, the naming conventions may be changed: Instead of appending -Wrapper to each wrapper object, an appendix like -TO (which stands for Transfer Object) may be appended to each business object and wrappers left without appendices. In that case, Album would be the client-side wrapper for the transfer object AlbumTO.

* * *

Client-side wrapper objects strike a balance between implementing business logic in business objects and creating static helper methods. Creating a class hierarchy of wrapper objects makes it possible to employ object-oriented concepts such as polymorphism while keeping this code local to the client application.

As a downside of this approach, the design of wrapper objects is more complex, in particular regarding associations. Creating dedicated wrapper objects leads to a proliferation of both classes at compile-time and objects at runtime. Wrapper objects are closely coupled to business objects. If the implementation of a business object changes, the wrapper object often also needs to be changed at once.

Typically, the code of data transfer objects is generated. Because the structure of wrapper objects follows static rules, the code for wrapper objects may be generated as well. In that case, a sophisticated code generation tool is required to mix both generated and hand-written code.

**Con-
sequences**

Freely Deployable Logic

Deploy the same code for business logic on the client-side as on the server-side.

Business logic that is needed on the client-side is needed on the server-side as well.

It is important to decide late where to deploy business logic.

* * *

It may not be desirable to implement any kind of business logic locally in client applications. However, some business logic that exists on the server-side may be useful on the client-side as well, for example for the validation of business data.

If business logic is left on the server-side only, performance problems are likely to appear; if existing business logic is re-implemented on the client-side, code redundancies may arise.

Therefore:

Implement business logic in a way that makes the logic freely deployable both locally on the client-side and on the server-side.

Employ a light-weight container or a library that encapsulates the business logic from the actual environment in which the business logic is deployed. Depending on the actual product, either use a standard programming language and declaratively deploy the code or use declarative programming to implement the business logic.

The key to implementing this pattern is the ability to run the container or library both on the client-side and server-side. This must be made possible only once. Then it can be decided freely and late for each part of the business logic where to deploy it.

In case of a container, business logic may be implemented without explicit references to the technical infrastructure that is available only on the server-side. The Spring Application Framework ([Spring]) is an example for a framework that supports this approach.

In case of a library for declarative programming, such a library interprets business logic that has been written, for example, in an XML dialect. The Jakarta Commons Validator project ([Validator]) supports the validation of business data in such a fashion.

Suppose, a Song instance must always have a title that must not exceed 40 characters. The following XML code describes this rule, using Apache Validator:

```
<formset>
  <form name="SongBean">
    <field property="title" depends="required,maxlength">
      <arg0 key="Song.title"/>
      <arg1 name="maxlength" key="{var:maxlength}"/>
      <var>
        <var-name>maxlength</var-name>
        <var-value>40</var-value>
      </var>
    </field>
  </form>
</formset>
```

**Additional
Context**

**Additional
Forces**

Solution

**Imple-
mentation**

This kind of business logic may be deployed both in a web server to validate user input from a web page and in a server component that checks business data before it is stored in a database.

* * *

The ability to deploy business logic freely gives the opportunity to decide late where to deploy the business logic. This leaves room, for example, for late performance optimizations without the need to change the system architecture.

**Con-
sequences**

If the same business logic is needed on both the client-side and server-side, source code redundancies are avoided. Deploying the same source code in several environments ensures that the business logic behaves the same in all cases.

The downside of this approach is the closely coupled development process of both client and server components. Furthermore, it may be difficult to find an appropriate library or framework to be used on both client-side and server-side that satisfies the particular project needs.

Discussion

The four patterns described in this paper do not exist in total isolation. Rather, some combinations and transitions are possible.

In reality, two completely disjunct sets of business logic for the client-side and the server-side are rarely present simultaneously – some logic often overlaps. It may be preferable, however, to implement `CLIENT-SIDE WRAPPER OBJECTS` in order to have separate sets of objects that each contain the business logic for either side.

In this case, business logic that is commonly needed may be implemented as `FREELY DEPLOYABLE LOGIC`. On the client-side, the wrapper objects may be extended by methods that delegate the calls to the freely deployable logic. As a result, the client application still only recognizes and uses the wrapper objects while code redundancies are avoided.

If client applications contain `CLIENT-SIDE HELPER METHODS`, it may become evident that some of these methods are also needed on the server-side. If it is possible to also implement `FREELY DEPLOYABLE LOGIC`, some individual methods may be re-implement as freely deployable logic and both approaches mixed.

In small projects, `RICH BUSINESS OBJECTS` may be a good solution to start as they are easy to use and dependencies between the development of the client-side and the server-side do not matter that much. If such projects grow bigger, however, a switch to `CLIENT-SIDE WRAPPER OBJECTS` may be desired.

For this transition, basic wrapper objects without any business logic of their own must be created first. For each rich business object method, a delegating method must be added to the wrapper object and all calls to the rich business object be refactored to call the wrapper object. Bit by bit, the actual business logic is then moved from the rich business objects to the wrapper objects until all client-side business logic has been removed from the business objects.

Apart from these considerations, some common consequences have not yet been mentioned.

In many applications, business data can only be modified with paying regard to side effects. This is one of the main reasons why a client application must not change any data directly but instead call appropriate business logic on the server-side. Neither pattern on its own prevents the implementation of business logic on the client-side that modifies business data.

Nevertheless, the strict application any of the patterns ensures the coherence of a client-side business logic layer. If business logic is not spread over many places, the reviewing of client-side business logic for the uncovering of violations is simplified.

Security must be enforced on the server-side. The constraints on business data on the client-side must be checked, for example when a user has entered data, to increase the user-friendliness of the system. These checks still need to be applied on the server-side as well because a server application can not rely on the correctness of a client application. Implementing `FREELY DEPLOYABLE LOGIC` and calling it on both sides decreases the risk of code redundancies.

Acknowledgements

I'd like to thank Didi Schütz who shepherded this paper for the EuroPLOP 2006 pattern conference. His thoughts and remarks have substantially helped me to get the paper in shape. I'd also like to thank Lutz Hankewitz and Dominik Dunekamp for their reviews and feedback.

References

- Alur, D, J. Crupi, and D. Malks. 2003. *Core J2EE Patterns*. Second Edition. Prentice Hall
- Evans, E. 2004. *Domain-Driven Design*. Addison Wesley
- Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns*. Addison Wesley
- Marinescu, F. 2002. *EJB Design Patterns*. John Wiley & Sons
- Riel, A. 1996. *Object-Oriented Design Heuristics*. Addison Wesley
- Spring. *Spring Application Framework*.
Available online: <http://www.springframework.org/>
- Validator. *Jakarta Commons Validator Project*.
Available online: <http://jakarta.apache.org/commons/validator/>
- Völter, M., J. Bettin. 2004. *Patterns for Model-Driven Development*.
In: Proceeding of EuroPLOP 2004.
Available online: <http://www.voelter.de/data/pub/MDDPatterns.pdf>