Business Logic in the Presentation Layer

Design Patterns on the Implementation of Business Logic on the Client-Side

Tim Wellhausen

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

Jan 05, 2007

Abstract: As a general rule of thumb, business logic in a multi-layered information system should be implemented in a dedicated domain layer and be separated from the presentation layer. 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 the presentation layer.

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

Introduction

In an information system, *business logic* represents the rules and activities of a business domain, whereas a corresponding *business domain model* defines the associations and properties of *business data*[1]. 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 from the user interface and the technical infrastructure. 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 the presentation layer of client applications. To make a client application convenient to use, some information about the business domain has to be incorporated into the presentation layer.

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 ([Evans2004]) 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 an application server whereas the presentation layer is part of a web or rich-client application. A common architecture for a web application is shown in Fig. 1.

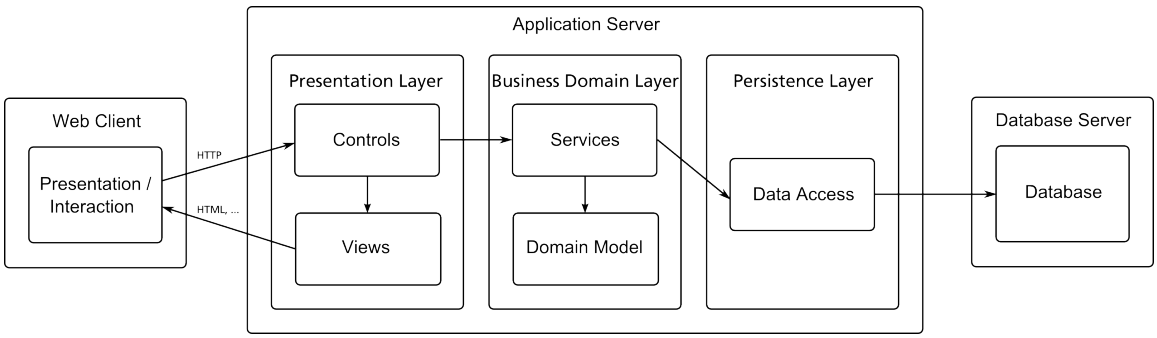

Fig. 1: Architecture of a Web Application

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 web client interacts with the presentation layer that in turn calls the business domain layer for all business-oriented tasks. The loading and saving of business data is delegated to a persistence layer that communicates with the database server.

In contrast, a common architecture for a rich-client application is shown in Fig. 2. The presentation layer is now locally deployed on the machine of a user. The communication between the presentation layer and the domain layer needs to cross the system boundaries, which involves network transfer.

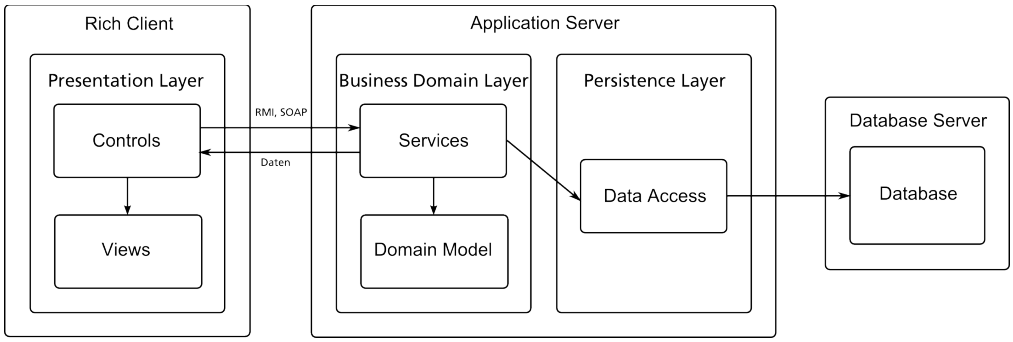

Fig. 2: Architecture of a Rich-Client Application

In both cases, business data should only be modified by the business domain layer to ensure the consistency and security of the data. Consistency checks should be performed by a central component to detect, for example, simultaneous changes to the same business object. Security must be enforced on the server-side because a server application must not rely on the correctness of a client application. Constraints on the valid values of business data should also be checked on the client-side, however, for example when a user has just entered data, to increase the user-friendliness of a system.

The specific structure of a business domain layer varies tremendously. Typically, the business domain layer consists of some kind of services (in [Fowler2002] they are called Transaction Scripts) and a Domain Model (also see [Fowler2002]).

These are some typical variations:

A simple domain model may contain business objects with primitive values only. In this case, the columns of a database table directly correspond to the fields of a business object. 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 connected business objects and support the usage of polymorphism to create business object class hierarchies. The presentation layer may then retrieve a fully expanded graph of business objects at once.

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 procedural, i.e. in services that load, modify, and store business objects.

Whether the presentation layer uses the same business objects as are used on the server-side may vary as well. If business objects are not sent to the presentation layer, typically, Data Transfer Objects ([Marinescu2002]) are exchanged between the server-side and the client-side.

* * *

Within this context, the following problem often needs to be addressed:

How do you implement business logic that is primarily needed by the presentation layer?

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:

Logic that is needed on the client-side often is 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. 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, business logic is strictly kept on the server-side, other problems may arise. Some logic may be relevant only to the presentation layer. If such a logic requires frequent modifications during the development 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 presentation layer may evoke security issues if modifications to business data are not checked on the server-side as well. Malicious users may modify a client application to let them ignore security checks.

In a multi-user environment, the presentation layer may not be able to enforce the integrity of business data, in particular in case of rich-client applications. Only a centralized server component is able to handle cases of synchronous user actions.

* * *

The patterns that are shown in Fig. 3 resolve these forces. Note that the patterns do not complement each other; they are rather alternatives for solving a common problem.

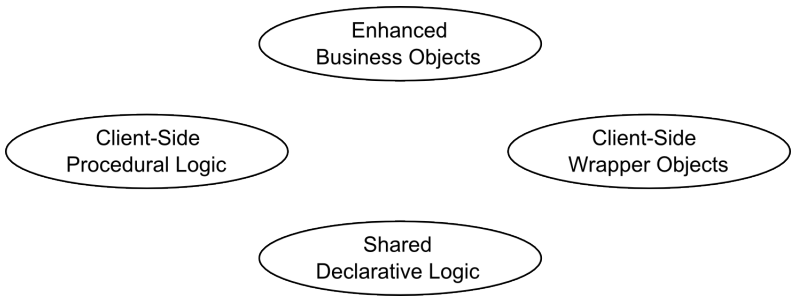

Fig. 3: The Patterns

You may implement client-side business logic in Enhanced Business Objects as it is the object-oriented way of coupling data and associated behavior.

If the presentation layer only receives pure data transfer objects and a simple solution is essential, you may implement business logic as Client-Side Procedural Logic.

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 the presentation layer and, at the same time, a decision regarding where to deploy the logic may be desired at a later time. Then you should consider Shared Declarative 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 as a rich-client application and connected to a server from which it fetches play lists and retrieves information about albums. The server-side business domain layer for this example application is depicted in Fig. 4. Note that although the running example is illustrated in Java, the patterns are not restricted to Java.

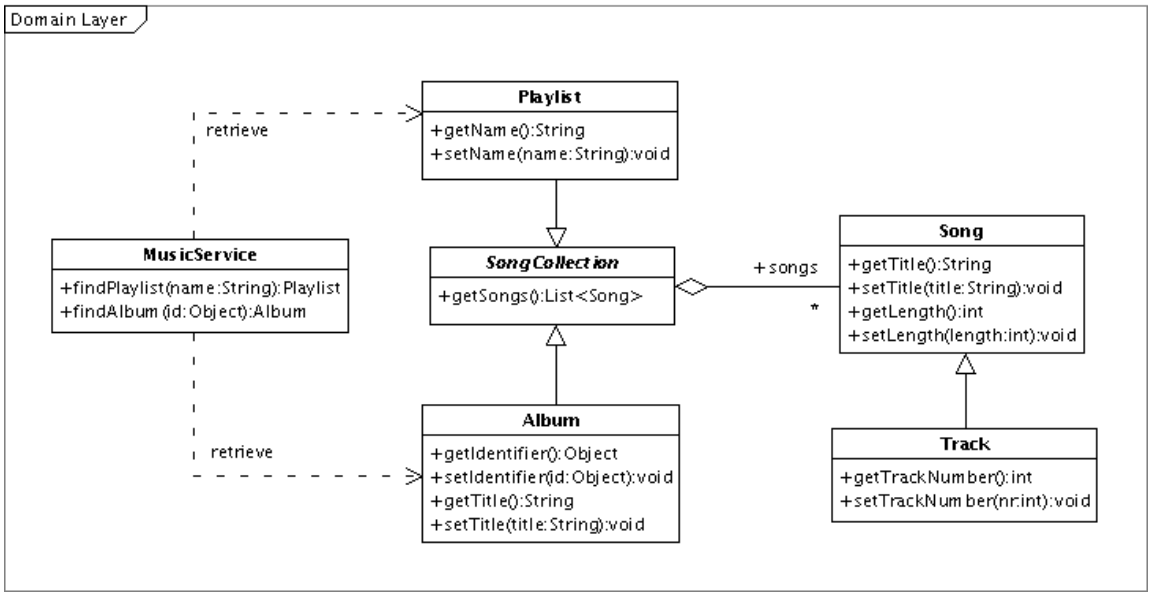

Fig. 4: Server-side domain model of a music player

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 keep the track number. An `Album` has a unique identifier and a title, a `Playlist` a name.

Moreover, suppose the following requirements apply 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.

Enhanced Business Objects

Implement client-side business logic in the business objects themselves and send these objects to the presentation layer.

The information system to build is a web application or the business domain model is straightforward and there are few complex requirements on the presentation layer. Additionally, the individual developers are mostly working on both the presentation and domain layers at once.

* * *

How do you implement business logic for the presentation layer if low overhead is important and redundancies should be avoided?

A system that can be built by a small team of developers may need a straightforward solution to implement business logic in the presentation layer. In particular, the extra effort to create a suitable solution must be low.

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 ([Riel1996]). If data and behavior are separated, business logic may become splattered over many places, which makes it difficult to keep track of it.

Therefore:

Keep business logic in business objects. Enrich each business object with the business logic needed by the presentation layer.

Let the business domain layer return business objects to the presentation layer when requested so that the presentation layer can use them to update the appropriate views. Therefore, add the functionality requested by the presentation layer to the business objects. (see Fig. 5).

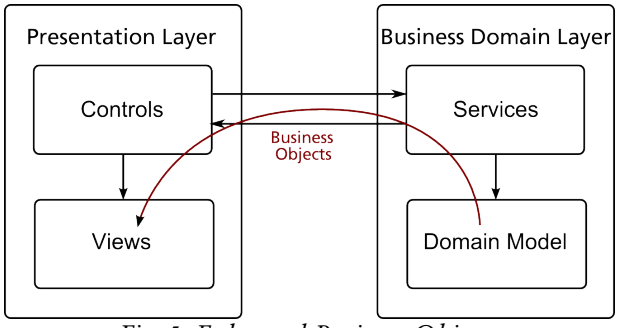

Fig. 5: Enhanced Business Objects

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

To resolve the requirements of the example application, the classes of the domain model are extended by client-side business logic, as shown in Fig. 6. Note that all methods that have been added are marked in blue color.

The class `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.

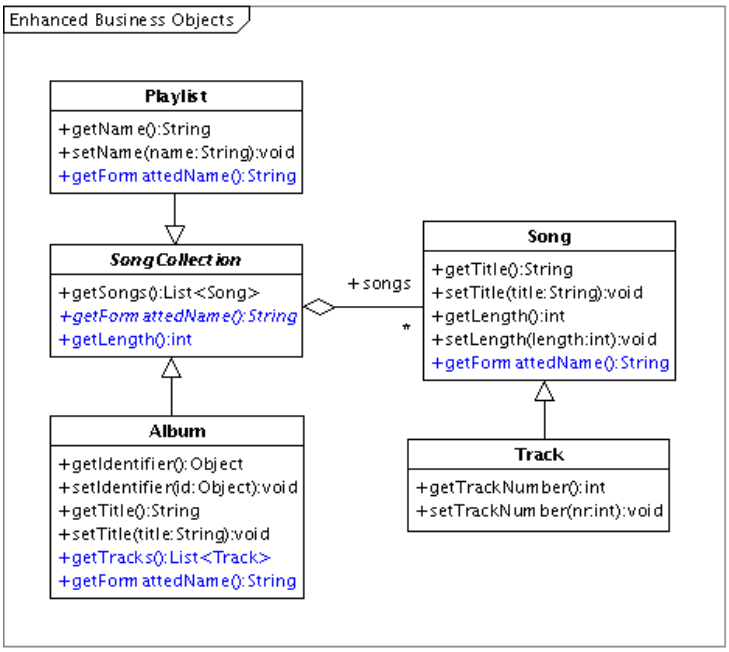

Fig. 6: Example resolved using Enhanced Business Objects

* * *

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 and therefore easier to locate and maintain.

There is almost no overhead involved in fulfilling the requirements of the presentation layer. Adding new methods to existing domain classes is easy and cheap.

On the other hand, 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. In particular, business objects that are transferred to a client may provide high-level operations that expose server-side business logic.

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 the presentation layer.

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+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 logic is added to business objects, 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 [Riel1996]).

Client-Side Procedural Logic

Implement business logic on the client-side in a procedural style.

The presentation layer and the domain layer either communicate by the exchange of pure data transfer objects or by the exchange of business objects that must not be enhanced for the presentation layer. Still, there are few complex requirements on the presentation layer.

* * *

How do you implement business logic in the presentation layer if low overhead is important and redundancies should be reduced?

If the presentation layer receives objects that do not contain the logic required for the presentation layer, this logic must be implemented locally in the presentation layer itself. If there is no policy how and where to implement client-side business logic, all kinds of code redundancies may result.

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.

Therefore:

Implement business logic in a procedural style and locate its implementation in a central place of the presentation layer.

It is important to organize the client-side procedural logic in such a way that it is not interwoven with the controls or views of the presentation layer. The individual procedures should receive the objects retrieved from the domain layer as parameters and return their results in the form needed by the presentation layer (see Fig. 7).

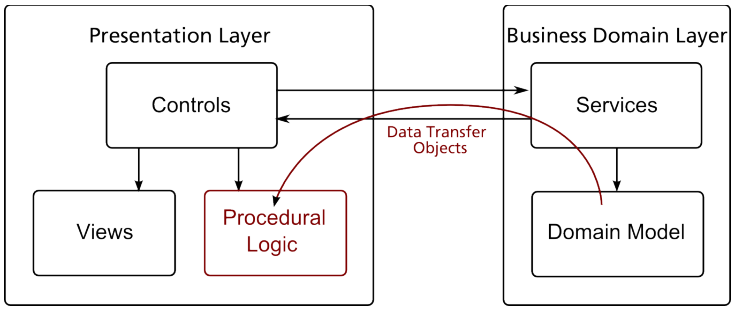

Fig. 7: Client-Side Procedural Logic

You could, for example, implement procedural logic in helper classes with static methods. In this case, you could create a helper class for each business object 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. You need to trade off between a proliferation of helper classes and the cost of introducing conditional statements.

To make the relationship 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.

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. 8 presents a possible implementation for the running example using procedural logic in form of static helper methods.

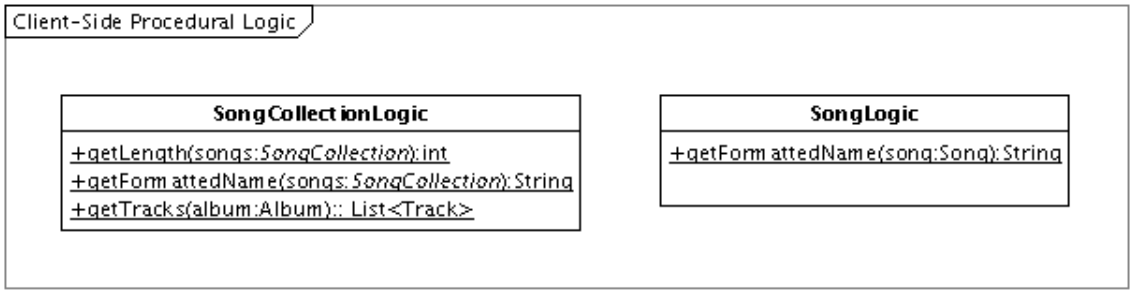

Fig. 8: Example resolved using Client-Side Procedural Logic

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 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.

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. The involved system are therefore only loosely coupled.

As a downside, business data is separated from its behavior. Procedural logic is often seen as a symptom of a badly designed object-oriented system. 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 on the client-side that contains the business logic.

The presentation layer and the domain layer communicate only by the exchange of pure data transfer objects. The business domain model is complex, and so are the requirements for the presentation layer.

* * *

How do you implement business logic in the presentation layer if it is important to keep the logic in one place and to take advantage of object-oriented programming?

Business logic can't be implemented in ENHANCED BUSINESS OBJECTS because the requirements on the client-side differ from the requirements on the server-side and the development process of the presentation layer and the domain layer would be bound too closely.

Implementing business logic in CLIENT-SIDE PROCEDUREAL LOGIC may cause business logic to deteriorate into conditional code that is difficult to understand and maintain. Object-oriented concepts may help to improve the structure of the code.

Therefore:

On the client-side, create a wrapper object for each data transfer object and add the business logic that is required by the presentation layer to the wrapper objects.

Each wrapper object receives a reference to the wrapped data transfer object upon its creation. The business logic in a wrapper object may then act upon the business data in its wrapped object and provide access to the wrapped business data. For each association of a wrapped business object, the wrapper object needs to maintain a new association to another wrapper object.

Although the presentation layer receives data transfer objects from the domain layer, the controls and views of the presentation layer are developed only against the wrapper objects (see Fig. 9).

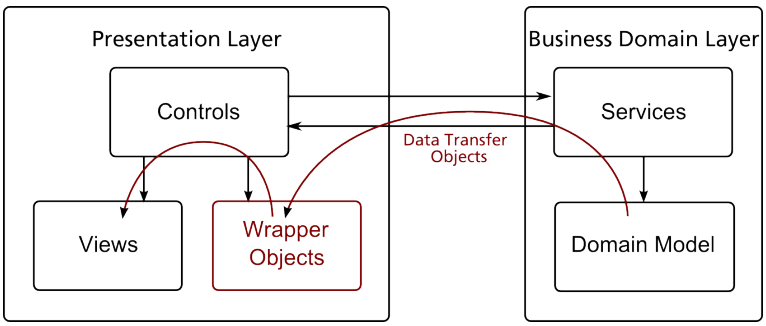

Fig. 9: Client-Side Wrapper Objects

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+2003]) to encapsulate this process.

To apply this pattern to the example application, a wrapper must be created for each business object, as shown in Fig. 10.[2] The upper half of the class diagram shows the business

2 There should actually be data transfer objects in the diagram; these are left for simplification.

objects from Fig. 4 (`Playlist` has been left out in this example). The lower half contains the corresponding wrappers: one for each business object (marked in blue color).

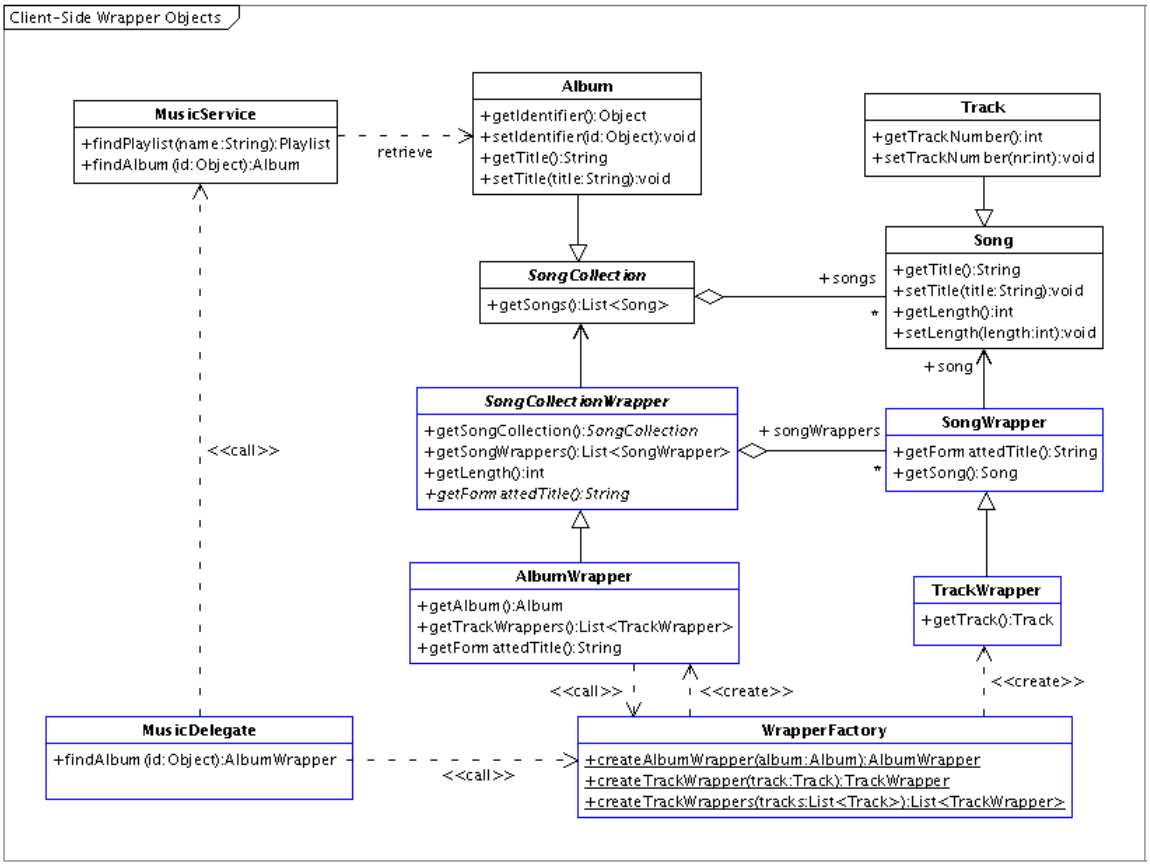

Fig. 10: Example resolved using Client-Side Wrapper Objects

The wrapper objects form a class hierarchy that corresponds to the hierarchy of the 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 ([Evans2004]). Both the business delegate objects that retrieve business objects from the server component and the wrappers themselves should call the wrapper factory for object 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 be 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 using 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 data transfer objects. If the implementation of a data transfer 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.

Shared Declarative Logic

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

Some kind of business logic is required both on the client-side and on the server-side. It may be important to decide late where to deploy this logic. It is possible to generalize the business logic.

* * *

How do you implement business logic that should be easily accessible and executable in different layers of a system?

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:

Apply an existing domain specific language or create a new domain specific language, implement the business logic declaratively, and make it available in any layer.

Business logic that is developed declaratively may easily be deployed in any layer of a system. It may be a runtime decision where to execute the logic. The key to this pattern is the ability to run a container or library that executes the business logic both on the client-side and server-side (see Fig. 11).

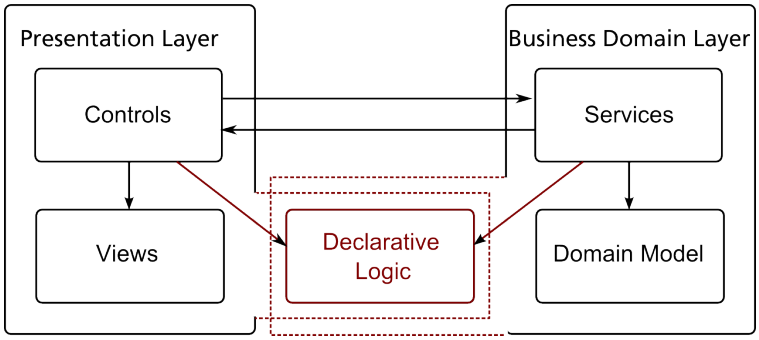

Fig. 11: Shared Declarative Logic

Typically, declarative business logic is written as text, for example in form of an XML dialect. The Jakarta Commons Validator project ([Validator]) supports the validation of business data in such a fashion.

To apply business data validation to the example application, 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>
```

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.

* * *

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 ability to decide late where to deploy the business logic leaves room for late performance optimizations without the need to change the system architecture.

The downside of this approach is the potentially huge upfront effort to create a domain language. There should be more reasons for such an intention than the desire to have a neat mechanism to execute business logic in the presentation layer.

Furthermore, it may be difficult to find an appropriate library or framework to be used on both the client-side and the server-side that satisfies the particular project needs.

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+2003] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns*. Second Edition. Prentice Hall, 2003

[Evans2004] E. Evans. *Domain-Driven Design*. Addison Wesley, 2004

[Fowler2003] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003

[Marinescu2002] F. Marinescu. *EJB Design Patterns*. John Wiley & Sons, 2002

[Riel1996] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996

[Validator] *Jakarta Commons Validator Project*. http://jakarta.apache.org/commons/validator/

[Völter+2004] M. Völter, J. Bettin. Patterns for Model-Driven Development. In *Proceedings of the 9th European Conference on Pattern Languages of Programming*, 2004 http://www.voelter.de/data/pub/MDDPatterns.pdf