IBM

developerWorks®

Patterns-based design and development for architects, Part 1: Using design patterns

(Got patterns?)

Arun Chhatpar (arunchhatpar@gmail.com)
Software Architect
Freelance Software Architect and Consultant

Skill Level: Intermediate

Date: 08 May 2007

Design patterns are one of the best ways to share design ideas. They give software architects and designers a tool, or a language, to capture their experiences by solving common recurring problems in a methodical way. In this two-part series you explore ways to use design patterns to solve your everyday design problems. This tutorial uses a railway reservation system case study to illustrate the design problems that can arise while architecting a system. Take this tutorial to learn how to use design patterns to solve problems and improve your own designs.

View more content in this series

Section 1. Before you start

This two-part series is for all programmers, architects, developers and technical enthusiasts who are interested in improving their software application designs. After finishing this series, you will be able to use best practices while choosing the right design patterns for solving specific problems.

About this series

This series demonstrates how to apply design patterns to architecture design problems using a railway reservation case study.

This tutorial introduces a railway reservation system and walks you through several design considerations that help determine where to use design patterns to improve the design and thus, the overall performance of the system.

Part 2 discusses nonfunctional requirements of the application, explaining why a software architect must take care of requirements that affect the performance, availability, scalability, and enhanceability of an application. It also touches on design considerations for disaster recovery and fail-back capabilities.

About this tutorial

This tutorial discusses design problems that a software architect can face while architecting a railway reservation system. It outlines different ways to solve a particular design problem using design patterns. The tutorial has three main sections, each starting with a brief discussion of design issues related to a specific part of the system:

- Interfacing with a legacy system
- Flexibility to use any database to maintain local data and transactions
- Using design patterns for a better user experience

Each section describes design solutions to the problem, first without design patterns and then with design patterns, including benefits and drawbacks of each approach.

Prerequisites

This tutorial assumes you are familiar with design patterns and with basic object-oriented concepts. Some understanding of UML is helpful, but not required. The sample code is written in Java®, but it's simple enough to be translated to the language of your choice.

You can download Java 5.0 if needed.

Section 2. The railway reservation case study

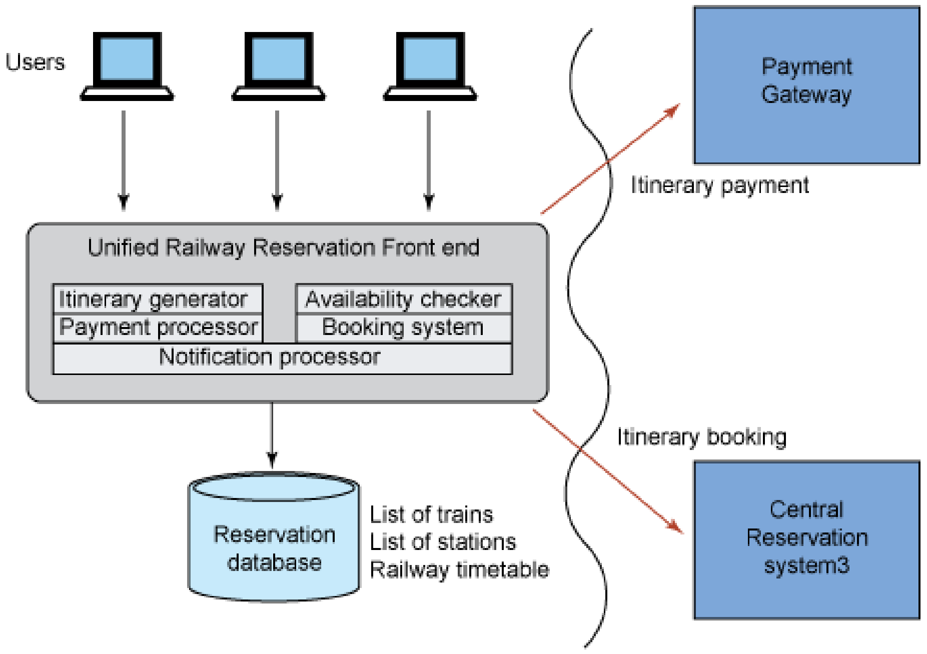
This section defines the case study involving a railway reservation system. It defines the business problem, with a brief introduction about each component involved in the system, an overview of the whole design, and use cases to outline the flow of execution. The information in this section is very important to understanding the rest of the tutorial and forms the basis for Part 2.

The business problem

First, a quick recap of *design patterns*. Design patterns are best defined as "a group of reusable assets that can help speed the process of software development." Design patterns represent solutions to problems that arise when developing software within a particular context. They capture the static and dynamic structure and collaboration among key components in software designs.

The application of various design patterns can benefit our case study. This tutorial takes a classic business problem of creating a Web front-end system to an existing application that uses legacy systems. The case study involves a hypothetical railway reservation system, shown in Figure 1.

Figure 1. The Unified Railway reservation system



The main components of this system are:

Central reservation system
: The legacy system that makes the actual reservations. It could have been written in COBOL, running on a mainframe system. All that matters for our tutorial is that it's the main system our front end needs to talk to for making reservations.

Payment gateway
: An external system that validates online purchase transactions.

Web container
: Unified Railway Reservation front end (This is the component we'll be talking about the most in this tutorial. It provides a unified Web interface to Web users.)

Local cache database
: Will hold a local copy of list of trains, stations, and a timetable between destinations. You'll learn about the rationale and advantages of having this local database.

Web users
: Users of the system (actual users or travel agents) adapting to the new interface.

Design overview

Almost every country has long-distance trains to commute from place to place. Railway departments have an electronic reservation system to make queries about schedules and prices, and to make bookings. The existing front end to this reservation system is their proprietary terminals, also called the green screens, which are mainly used by travel agents.

To explain an important design problem, it is assumed that the central reservation system team has exposed a SOAP interface that will allow a Web application to interface with the existing system. The local cache database shown in Figure 1 maintains a list of stations, trains, and schedules that are updated periodically from the central reservation system. It provides quick access to all this data, which is less prone to frequent changes.

This tutorial discusses problems that arise while designing an application that acts as a unified front end to the central reservation system. The application will be Web-based and will allow users to select their itinerary. After reviewing the itinerary, users must submit for confirmation and the front end will send a request to the underlying reservation system to confirm the tickets. It will also send necessary notifications, such as a confirmation e-mail and an option to print the itinerary.

Use cases

The use cases correlate to the functions the end user is expecting. They are helpful in understanding the flow of execution in the system. The use cases are:

Select itinerary

On the Trains Inquiry form, users will enter the following information:

- Starting station
- Destination station
- Date
- Number of passengers

The system should list the trains that run between the mentioned destinations on the specified date.

Check availability

From the list of trains, users will select one and check for availability of number of seats on that train. Once they make a selection, users can go ahead and book the tickets.

Book tickets

Users will specify the name and age of each passenger, and submit the data to book tickets. For this operation, the front end application will send a request to the central railway reservation system.

The central system will issue a unique booking reference number that will be valid for 5 minutes. If payment is not made within 5 minutes, the transaction will be invalidated and the seats that were allotted earlier will be freed for other bookings.

Make payment
User will make the payment electronically using a payment gateway. For this tutorial, the payment gateway is just an external system that validates the payment information.

E-ticket & notification
Upon successful payment, the system will generate an e-ticket for printing. It will also send e-mail notification to the e-mail address specified by the user.

There are many design areas in this business case that require a good amount of thinking; the following sections cover some of them. Each section introduces the design problem, suggests the common way to fix it, then shows a better and more reusable way to fix the same problem using design patterns.

Section 3. Interface to a legacy system

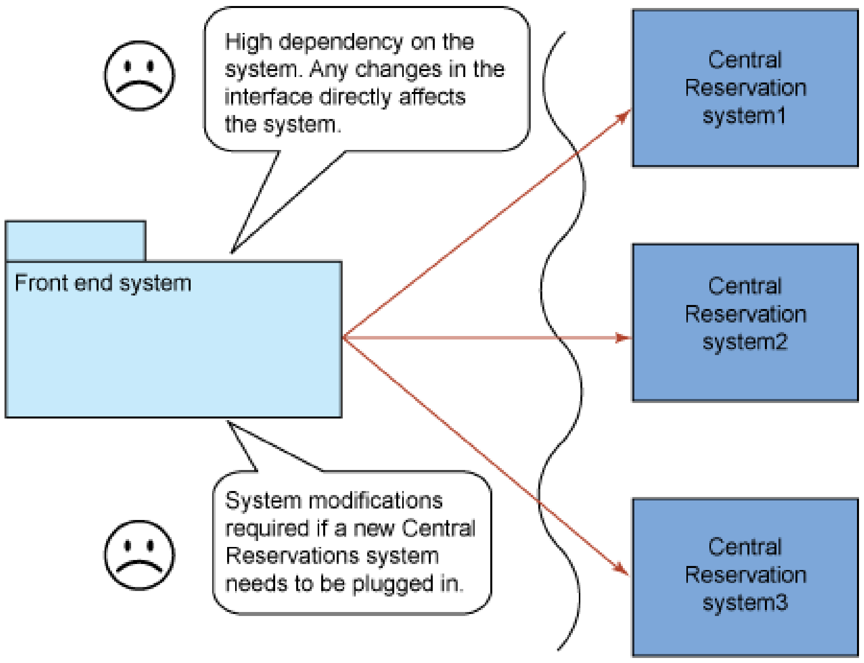
This section examines the front-end system that should plug into any railway reservation system that supports communication using services. There are a couple of ways to tackle this: the first is simple and direct, and the second makes use of design patterns.

Creating a solution without design patterns: no interface abstraction between components

Since the target front-end system is supposed to act as a unified interface for any railway reservation system, it should be able to plug into any such application. Every reservation system will have their proprietary request and response formats. We'll call these formats the *interface* of the system.

Essentially, each reservation system will have its own interface, which may or may not be the same as the one designed for our application. Our application will have to translate its interface to the interface of the underlying reservation system. Because the translation logic will be hard-coded in our implementation of the interface, it will have to be revamped if the underlying central reservation system changes. Figure 2 highlights this scenario.

Figure 2. Direct communication with legacy system


High dependency on the system. Any changes in the interface directly affects the system.
Front end system
System modifications required if a new Central Reservations system needs to be plugged in.
Central Reservation system1
Central Reservation system2
Central Reservation system3

Listing 1 encapsulates all of the logic to communicate with the legacy system.

Listing 1. Direct communication with legacy system

```
public class TicketBooker {
    public boolean bookTicket() {
        // Logic to interface with the legacy system
        return true;
    }
}
```

In the scenario in Figure 2, the system directly translates the requests to what the central reservation system understands. There are, however, drawbacks to this approach including:

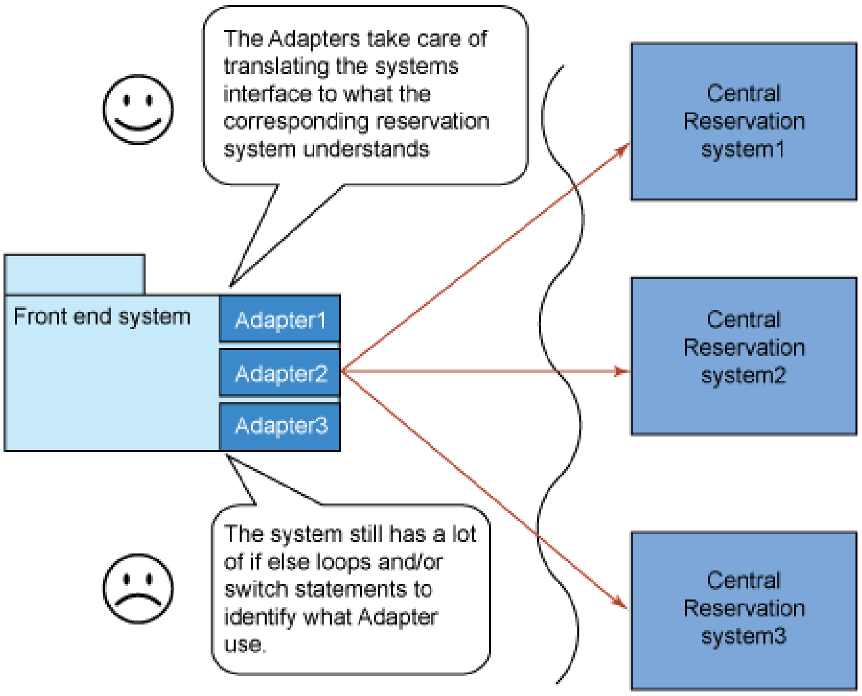
- The application is highly dependent on the interface. Because the translation logic is hard-wired in the interface, any change in the interface of the central reservation system directly impacts the front-end application. The impact may be in a single module, or spread across multiple modules.
- It becomes difficult to extend the system to communicate to any new reservation system. Again, the impact of the changes may be limited to a single module or spread across multiple modules.

This is the common way of taking care of communications with the legacy system. A better way to solve the problem is to localize the impact of interface changes by introducing a component that acts as a translator between the application interface and the interface of the reservation system, as discussed next.

Creating a solution using design patterns: the Adapter design pattern

This section explains how to solve the problem with design patterns. Figure 3 shows how to apply the Adapter design pattern as an intermediate layer for translation to our example.

Figure 3. Communication with adapters


The Adapters take care of translating the systems interface to what the corresponding reservation system understands
Front end system
Adapter1
Adapter2
Adapter3
The system still has a lot of if else loops and/or switch statements to identify what Adapter use.
Central Reservation system1
Central Reservation system2
Central Reservation system3

Listing 2 shows the same `TicketBooker` class shown in Listing 1, but this time it's interfacing with the legacy system through the adapter. All the `TicketBooker` class needs to know about is the `IRSAdapter` interface.

Listing 2. `TicketBooker` uses an adapter to communicate with legacy system

```
public class TicketBooker {
    public boolean bookTicket(Itinerary i) {
        // Use the Adapter to communicate with the legacy system
        IRSAdapter adapter = new IRSAdapter();
        adapter.bookTicket(i);
        return true;
    }
}
```

In this scenario, the system delegates the task of translation from one interface to another to the specific adapters. There are benefits and drawbacks to this approach, however.

Benefits

By introducing the adapters to translate the application interface to the reservation system interface and vice versa, the impact of changes in the reservation system interface are localized. The Adapter design pattern provides a way for a client to use an object whose interface is different from the one expected by the client, without having to modify either. This design pattern allows classes to work together that previously couldn't because of incompatible interfaces.

Drawbacks

There are a few drawbacks using this approach:

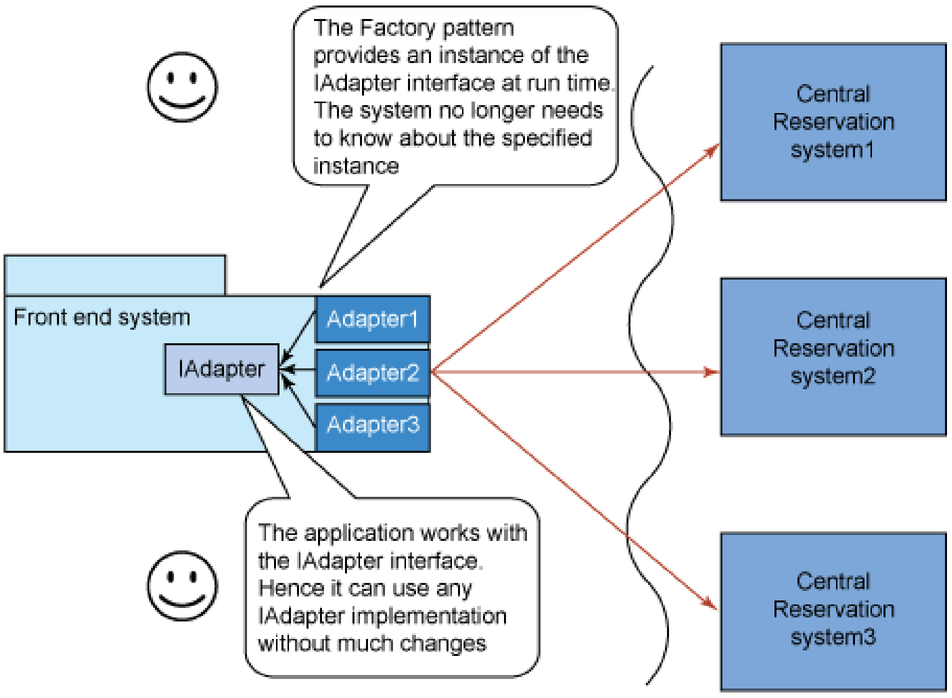
- The system still has to identify which adapter instance to use.
- The adapters are exposed to the communication protocols required to communicate with the reservation systems.

The drawbacks of the Adapter pattern leads to our next design pattern. To make the system independent of the adapter instance to be used at run time, you can implement the Factory design pattern.

Improved solution with more design patterns: Factory of adapters

The Factory design pattern defines an interface for creating an object at run time, but it leaves the decision of which class to instantiate to its subclasses. In other words, it defers instantiation to subclasses. Figure 4 shows how to make the best use of this design pattern.

Figure 4. Factory for choosing the right adapter


The Factory pattern provides an instance of the IAdapter interface at run time. The system no longer needs to know about the specified instance
Front end system
IAdapter
Adapter1
Adapter2
Adapter3
Central Reservation system1
Central Reservation system2
Central Reservation system3
The application works with the IAdapter interface. Hence it can use any IAdapter implementation without much changes

Listing 3 shows the use of the Factory design pattern.

Listing 3. `TicketBooker` class uses `AdapterFactory` to get the right adapter to communicate with legacy system

```
public class TicketBooker {
    private String legacySystemName;
    public TicketBooker(String legacySystemName){
        this.legacySystemName = legacySystemName;
    }

    public boolean bookTicket(Itinerary i) {
        // Use the Factory to get the right Adapter
        // Then use the Adapter to communicate with the legacy system
        IAdapter adapter = AdapterFactory.getAdapter(legacySystemName);
        adapter.bookTicket(i);
        return true;
    }
}
```

Some benefits of using the Factory pattern in our application are:

- The front-end system will communicate with the interface `IAdapter` without even bothering about the actual adapter instance.
- The actual `Adapter` instance that the system will be using at run time will be provided by the Factory pattern.

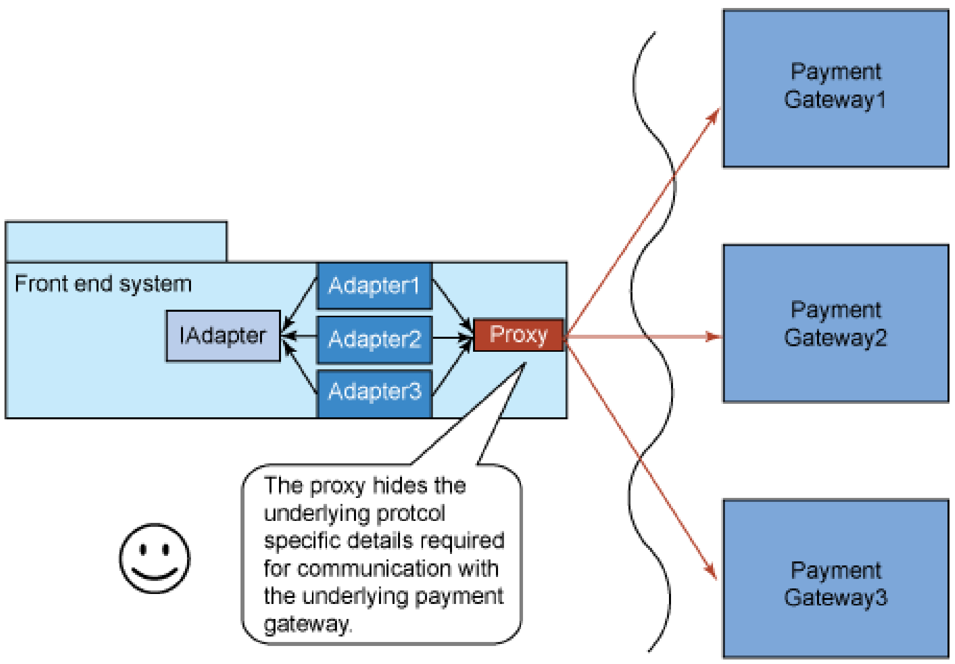
This took care of another design issue! But there is one important issue that should not be overlooked while designing a system: direct dependency on a communication protocol. It would be best if our front-end application is abstracted from it as much as possible. The next design pattern takes care of this problem.

Enhancing your design using patterns: Proxy design pattern

To abstract the application from the communication protocol, you can introduce a proxy object that hides the underlying communication protocol used to talk to the central reservation system. As shown in Figure 5, the proxy will:

- Take the transformed request from the adapter.
- Wrap the request with protocol specific details.
- Perform the communication.
- Trim the response by removing the protocol specific details.
- Send the response back to the adapter.

Figure 5. Proxies to hide communication details


Front end system
IAdapter
Adapter1
Adapter2
Adapter3
Proxy
Payment Gateway1
Payment Gateway2
Payment Gateway3
The proxy hides the underlying protcol specific details required for communication with the underlying payment gateway.

The design patterns just discussed take care of some of the legacy system dependency issues, but the next problem on our plate is related to database communications.

Section 4. Design patterns for database interactions

A goal in the scenario is to have the flexibility to use any database to maintain local data. The front-end system in the case study has to maintain a local record of user profiles and their transactions. The database used to maintain this local data can vary on implementation, and the design should make sure that it is not hard-wired to talk to a particular instance or type of database. The following sections explore the issues related to database connections.

Let's assume that the system has to maintain the following local data elements:

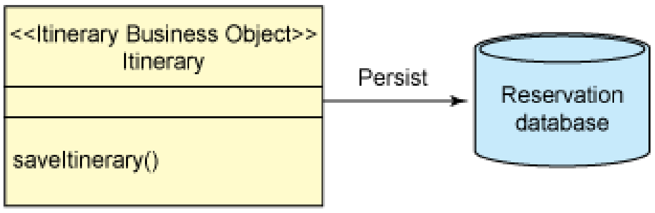
- User profile, along with their preferences
- Itinerary
- Payment information
- Transaction history

The design should be extensible enough to adapt to any database without recompiling the code. We'll start with the most basic implementation, then refine it by using some patterns.

Solution without design patterns: direct database communication

The most basic implementation will have the system classes directly communicate with the underlying database. The classes that implement the business logic are also responsible for all database communications, as shown in Figure 6.

Figure 6. Direct database communication



Listing 4 shows the code snippet that encapsulates all of the database communication logic in the `Itinerary` class. The `saveItinerary` method will have code to read the database connection parameter, connect to the database, execute database-related actions, and then disconnect from the database server. This would be hard-wired for a particular database, such as Oracle. If system managers decide to go with a different database in the near future, then you will have to go into the code and change not just this reference, but each such reference where database calls are being made.

Listing 4. `saveItinerary` encapsulates all the logic to communicate with database

```
   public class Itinerary implements Serializable{
    // Attributes containing the details of Itinerary
    public void saveItinerary(){
        // Logic to write the itinerary details directly to the database.
    }
}
```

In this scenario, the business object itself is responsible to communicate with the underlying database.

There are some drawbacks:

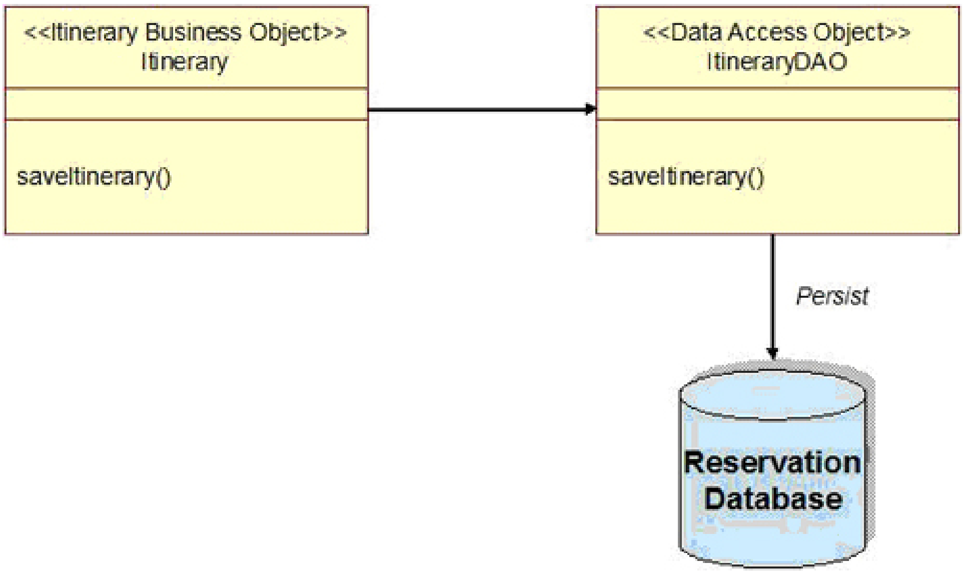
- There is a tight coupling between the business logic and the data access logic.
- If the underlying database is changed, the business object is affected.

It would be best to split the business logic implementation from a data access logic implementation to make the business object independent of data access logic. And that is exactly what we will do with our next design pattern.

Creating a solution using a design pattern: the Data Access Object pattern

You can use the Data Access Object (DAO) pattern to abstract and encapsulate all access to the data source. The DAO will manage the connection with the data source to obtain and store data. Figure 7 shows the use of DAO in the design.

Figure 7. Direct database communication



As shown in Listing 5, the application class does not need to know about the actual database it's talking to; it just needs to know the methods of the DAO interface. The DAO implementation class will have the database-specific logic. Now if you want to move to a different database, it's just a matter of writing another implementation of the DAO interface that will contain all the code to talk to that new database. Listing 5 shows a simplified code snippet to call the DAO implementation written for the `Itinerary` class.

Listing 5. Using DAO to remove database communication dependencies

```
public class Itinerary implements Serializable{
    // Attributes containing the details of Itinerary
    public void saveItinerary(){
        // Leave the underlying detail of database communication to the DAO
        ItineraryDAO dao = new ItineraryDAO();
        dao.saveItinerary();
    }
}
```

In this scenario, the business logic is separated from the data access and persistence logic. There are benefits and drawbacks.

Benefits

- The business logic is independent of the data access and persistence logic.

- It's easier to add or change the database implementation without really affecting the actual application. You just have to write a new implementation of DAO for that new database.

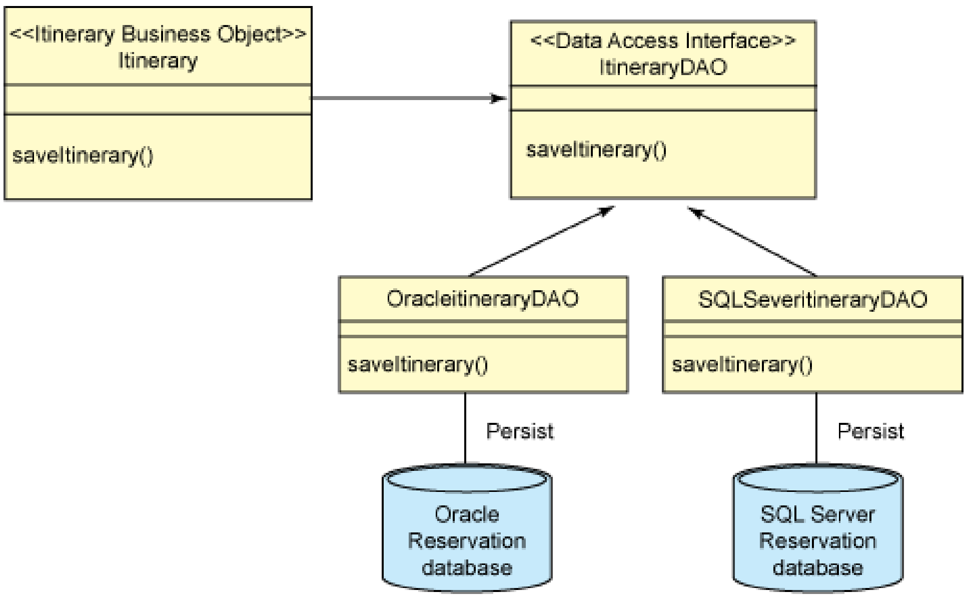
Drawbacks

The data access and persistence logic is dependent on the type of database. The application has to know about the class that implements the database functions. Remember the Factory pattern used with the adapter? It would come in handy here as well.

Enhancements with more patterns: using a Factory of Data Access Objects

It's simple. Just make the `ItineraryDAO` work on an interface and let an `Itinerary` factory provide the right `ItineraryDAO` instance at run time. Figure 8 outlines the concept.

Figure 8. Factory to identify the right DAO



Listing 6 shows how to use the Factory design pattern to provide the flexibility and extensibility your application needs for database communications.

Listing 6. Factory design pattern in conjunction with DAO pattern

```
public class Itinerary implements Serializable{
    // Attributes containing the details of Itinerary
    public void saveItinerary(){
        // Get the right DAO from the factory and
        // Leave the underlying detail of database communication to the DAO
        ItineraryDAO dao = DAOFactory.getDAO("ORACLE");
        dao.saveItinerary();
    }
}
```

That takes care of some of the database dependency issues, but another important consideration is performance.

Section 5. Using design patterns for a better user experience

So far in this tutorial you've learned how to remove dependencies between different components of your design by either introducing a wrapper between the two components or by introducing a helper class such as a DAO. These are *refactoring* issues, in software designing terms.

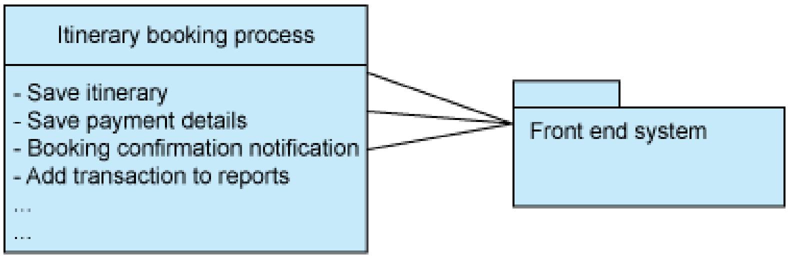
This section touches on another important design consideration: performance. Performance directly relates to user experience in most software applications, so it's a very important issue that should be well designed.

In our front-end application design, there are few actions that take place after the booking has been made. The system just needs to send e-mail confirmations. But as we all know, this might not be the case for very long. The very next requirement could be to add everything about this transaction into some type of auditing system, which can be then used to generate reports. There are ways to take care of such issues, as discussed next.

Solution without design patterns: synchronous processing

There can be several steps to finishing a particular process, and it is quite possible that some of the steps don't need to be synchronous. For example, the itinerary booking in our application involves taking the steps shown, as in Figure 9.

Figure 9. Synchronous processing


Itinerary booking process
- Save itinerary
- Save payment details
- Booking confirmation notification
- Add transaction to reports
...
...
Front end system

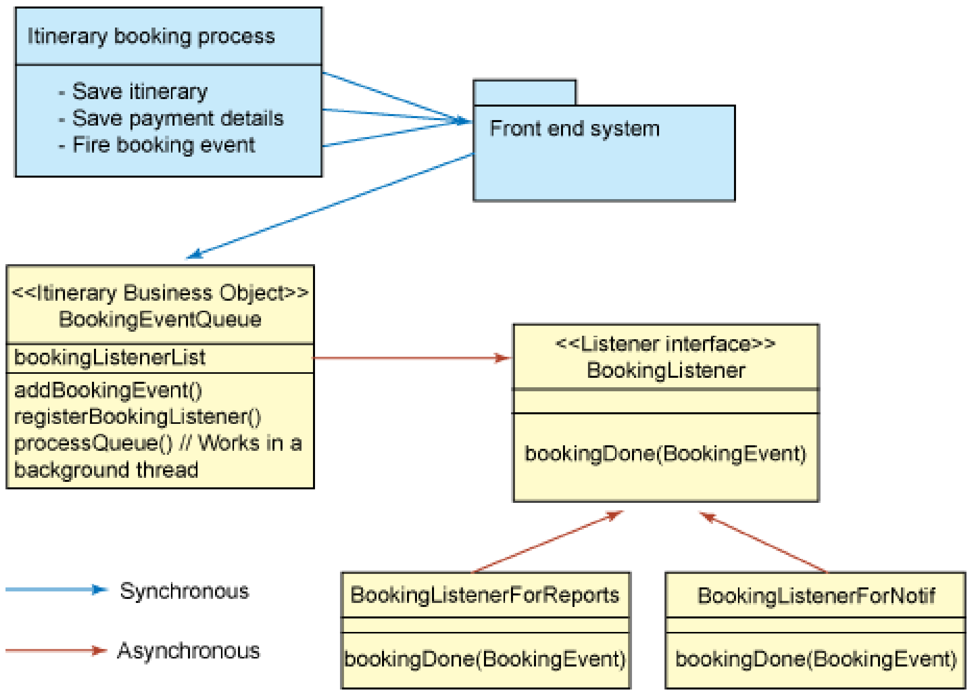
As the process flow indicates, the booking process is not complete until all of the background tasks are complete, and the user has to wait for a response. The situation becomes even worse as more and more tasks or steps are added for more functions. It's not good for the user experience to wait until all background tasks are complete. As far as users are concerned, the payment is done and that's all they care about.

Why can't the system do the notification processing in the background? That's exactly what will be done in the next scenario, by making the background processing task asynchronous.

Solution with design patterns: asynchronous processing

You can fix the notification processing problem by making the noncritical tasks asynchronous, as shown in Figure 10.

Figure 10. Asynchronous processing


Itinerary booking process
- Save itinerary
- Save payment details
- Fire booking event
Front end system
<<Itinerary Business Object>> BookingEventQueue
bookingListenerList
addBookingEvent()
registerBookingListener()
processQueue() // Works in a background thread
<<Listener interface>> BookingListener
bookingDone(BookingEvent)
BookingListenerForReports
bookingDone(BookingEvent)
BookingListenerForNotif
bookingDone(BookingEvent)
Synchronous
Asynchronous

This scenario uses the Listener pattern along with an event dispatcher. Once the booking is done and the payment made, the system fires a booking event and adds the event to the booking event queue. Once the event is added to the queue, the control returns to the user and they get to see the booking details.

The booking event queue keeps processing the events added in the queue in a separate thread. The queue can have an `n` number of `BookingListener` implementations registered and listening for booking events. At each run, the event processor thread picks every event in the queue and dispatches it to every registered Listener. The event is removed from the queue only when all listeners have successfully completed the processing.

Benefits

The benefits are:

- Noncritical tasks are made asynchronous to the main processing.

- Design is scalable, allowing more tasks to be added in the future.
- Improved user experience.

Section 6. Summary

Using a case study of a railway reservation system, this tutorial provides an introduction to using design patterns. Each design problem has multiple solutions; the aim is to make your design scalable and reusable. You also want your application to be flexible and fast. Patterns such as Factory, Adapter, and Data Access Objects, among others, can help you make your application flexible enough to plug into disparate systems without too much trouble. Asynchronous processing of noncritical events helps create a better user experience, too. Some distinct advantages, such as providing a rich vocabulary, make design patterns a valuable tool for designing object-oriented systems.

Stay tuned for Part 2, which covers the nonfunctional considerations that an architect or designer should not miss. Availability requirements, scalability considerations, maintainability, and enhanceability requirements will be discussed. Part 2 also explains how to use existing software frameworks like Spring and Struts to achieve the most out of your design and architecture.

Resources

Learn

- "Web services architecture using MVC style" (developerWorks, Feb 2002) takes a look at how the MVC pattern can be applied to the call static or dynamic Web services.
- Read simple explanations of all 23 design patterns.
- "A Survey of Common Design Patterns" from Developer.com summarizes the 23 common design patterns.
- Read Designing Enterprise Applications with the J2EE™ Platform, Second Edition, which describes standard approaches to designing multitier enterprise applications with the Java™ 2 Platform, Enterprise Edition.
- The Patterns Library has information about all aspects of software patterns and pattern languages.
- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides (Addison-Wesley, 1995) is considered an authority on design patterns. If you are new to design patterns, this is where you should start.
- Core J2EE Patterns: Best Practices and Design Strategies by Alur, Crupi, and Malks (Prentice Hall, 2001) is a catalog of patterns for the design and architecture of multi-tier enterprise J2EE applications.
- UML Distilled: A Brief Guide to the Standard Object Modeling Language by Martin Fowler with Kendall Scott (Addison-Wesley, 2000) is a very good resource for learning the essentials of UML.
- The Unified Modeling Language User Guide, by Grady Rumbaugh, James Jacobson, Ivar Booch (Addison-Wesley, 1998) is a reference guide to learn more about UML and design patterns.
- IBM Patterns for e-business explains this group of reusable assets that can help speed the process of developing Web-based applications.
- IBM Pattern solutions shows you how to get started using patterns.

Get products and technologies

- Get the RSS feed for upcoming installments in this series.
- Download IBM product evaluation versions and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Participate in the discussion forum for this content.
- Visit the developerWorks Architecture zone for architecture related tutorials and articles.
- Open source: Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies, and use them with IBM products.

- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.
- Podcasts: Tune in and catch up with IBM technical experts.
- Check out developerWorks blogs and get involved in the developerWorks community.

About the author

Arun Chhatpar

Arun Chhatpar has more than nine years of significant experience in Java programming and client/server architectures, and is a Sun Certified enterprise software architect. He has been a lead designer and developer for NBCi, and is currently working as an independent software architect/consultant.