Extracted from:

# Interface-Oriented Design

# Chapter 3
# Interface Ingredients

Now that we've covered the basics of interfaces, it's time to examine the ingredients of interfaces. Almost every interface you employ or develop has a combination of these ingredients, so understanding them helps you appreciate the whole pie. In this chapter, we'll look at the spectrum of interfaces from data-oriented to service-oriented and cover the trade-offs in three distinct approaches to data-access interfaces.

You can always adapt an interface paradigm from one type to another to make it more amenable to your project, so we'll explore how to adapt a stateful interface to a stateless one. Then we'll look at transforming a textual interface into a programmatic one and creating an interface from a set of existing related methods.

## 3.1 Data Interfaces and Service Interfaces

There is a spectrum between data interfaces and service interfaces. We use the term *data interface* when the methods correspond to those in a class that contains mostly attributes. The methods in the interface typically set or retrieve the values of the attributes.[1] We use the term *service interface* for a module whose methods operate mostly on the parameters that are passed to it.

One example of a data interface is the classic Customer class. Customer usually has methods like

- set_name(String name)

[1]Data interfaces also correspond to JavaBeans or pretty much any class that is a wrapper around attributes with a bunch of getter/setters.

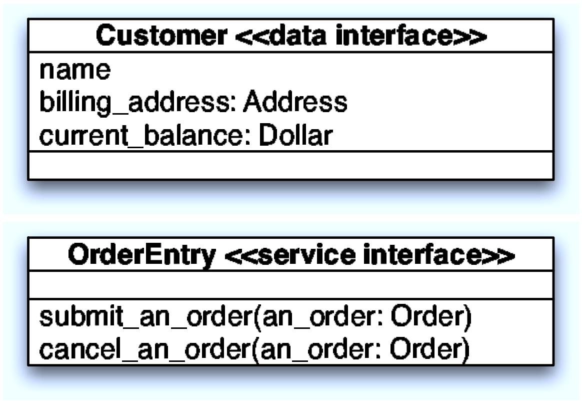
Customer <<data interface>>
name
billing_address: Address
current_balance: Dollar
OrderEntry <<service interface>>
submit_an_order(an_order: Order)
cancel_an_order(an_order: Order)

Figure 3.1: Data vs. service interface

- set_billing_address( Address billing_address)
- get_current_balance().

Each of these methods affects or uses an attribute in the class. Implementations of data interfaces have state, which consists of the set of values of all attributes in the class.

Service interfaces have methods that act on the parameters passed to them, rather than the attributes of the implementation, for example the methods submit_an_order(Order an_order) and cancel_an_order(Order an_order). Figure 3.1 shows how data interfaces have just attributes and service interfaces have just methods.

Service interface implementations usually have no attributes or only ones that are associated with providing the service, such as connection information that identifies where to submit an order or where to find the current price for a stock. Implementations of service interfaces may have no state, other than that of internal configuration values such as this connection information.

This data versus service interface comparison is not pure black and white, but rather a spectrum. An interface can range from a data transfer object (DTO), whose methods refer only to attributes of the object, to a command interface, which usually contains only service methods. We could move away from a pure data interface by adding methods to the Customer interface. We might add charge_an_amount(), which

## Entities, Control, Boundary

In *Object-Oriented Software Engineering*, Ivar Jacobsen introduced three stereotypes for objects: entity, boundary, and control. An *entity* depicts long-lived objects. *Boundary* objects communicate between the system and the actors (users and external systems). A *control* object represents behavior related to a specific use case. It communicates with boundary objects and entity objects to perform an operation.

These stereotypes relate to the data and service interfaces. Data interfaces correspond to the entity objects. The underlying data mechanism (e.g., database table or XML file) is opaque to the user of the entity object. An interface such as Pizza, which contains just the size and toppings, is an entity.

A boundary corresponds to a service interface. You push a button on a GUI or make a call to a method, and the underlying service is performed. The PizzaOrdering interface presented in Chapter 1 is a boundary interface.

A controller also corresponds to a service interface. Its methods are typically called by a boundary interface. It can embody business rules or services. A PizzaMaker that controls the making of the Pizza( ) could exist between the PizzaOrdering( ) and a Pizza( ). The PizzaMaker( ) would be a controller.

alters current_balance; mail_statement(), which mails the current_balance to the address; or is_credit_worthy(), which applies some business rules to determine whether to extend credit to the customer.

Let's take the PizzaOrdering interface in the first chapter and transform it into two interfaces on each end of the spectrum. First we make a pure DTO—a Pizza class containing just data on the pizza. For example:

```
class Pizza
    set_size(Size)
    set_topping(Topping)
    Size get_size()
    Topping [] get_toppings()
```

We now create a service interface that accepts a Pizza and places the order:

```
interface PizzaOrderer
    TimePeriod order_pizza(Pizza)
```

The method calling this interface first creates a Pizza and then passes the Pizza to order_pizza().

We can turn the Pizza class into a class endowed with more behavior, which is a guideline for object-oriented design. Let's add a method so that a Pizza orders itself:

```
class Pizza
    // as above, plus:
    order()
```

Pizza's order() method could call an implementation of the PizzaOrderer interface. One implementation could communicate the order over the telephone; another implementation could fax it or email it. The user of Pizza does not need to know about PizzaOrderer, unless they intend to change the implementation that order() uses.[2]

In Chapter 1, you ordered a pizza over the phone. If PizzaOrderer represented a phone-based system, before accessing order_pizza(), you need to call the shop. If we include that operation in this interface, it would look like this:

```
interface PizzaOrderer
    call_up()
    TimePeriod order_pizza(Pizza)
    hang_up()
```

Now PizzaOrderer represents a *service provider interface*, a variation of the service interface. A service provider adds methods to the interface that control the life cycle of the service provider. These methods are often called *initialize*, *start*, or *stop.* Java applets, servlets, and session beans are examples of service provider interfaces.

## 3.2 Data Access Interface Structures

You may run across different paradigms for interfaces that access data, so it's a good idea to appreciate the differences among them. An interface can provide sequential or random retrieval of data. Users can either pull data or have it pushed upon them.

[2]We explore ways to configure different implementations in Chapter 7.

## Sequential versus Random Retrieval

Data can be available in a sequential or random manner. For example, the Java FileInputStream class allows only sequential access, while RandomAccessFile allows access to the data in a file in any order.

The same dichotomy exists within collections and iterators. An iterator interface allows access to a single element in a collection at a particular time. Some styles of iterators, such as Java's Iterator or C++'s forward iterators, permit only one-way access. You have to start at the beginning of the collection and continue in one direction to the end. On the other hand, a vector or array index, or a C++ random-access iterator, allows random access to any element in the set. If you have data available with only sequential access and you want it to have random access, you can build an adapter. For example, you can create a vector and fill it with the elements from an iterator.

Other examples of sequential vs. random access are two Java classes for accessing the data in an XML file. The Simple API for XML (SAX) parser provides for sequential access to the XML elements; SAX does not keep the data in memory. On the other hand, the Document Object Model (DOM) allows random access. It creates an in-memory representation of the XML data. Note that a DOM parser can use a SAX parser to help create the memory representation. These two interfaces have corresponding advantages and disadvantages.

SAX: SEQUENTIAL ACCESS

  Advantage—requires less resources to parse the file

  Disadvantage—application cannot change the XML data

DOM: RANDOM ACCESS

  Advantage—application can change the XML data

  Disadvantage—requires memory to store the entire document

We'll revisit SAX and DOM in a little more detail in a later section.

## Pull and Push Interfaces

Interfaces move data in one of two ways: push or pull. You ask a pull-style interface—for example, a web browser—for data. Whenever you desire information, you type in a URL, and the information is returned. On the other hand, a push-style interface transfers data to you. An email subscription is a push-style interface. Your mail program receives information whenever the mail subscription program sends new mail. You don't ask for it; you just get it.

You can use either a pull style or a push style when going through a collection.[3] An example of a pull style is the typical iteration through a list or array:

```
Item an_item
for_each an_item in a_list
    {
    an_item.print()
    }
```

For each element in a_list, the print() method is explicitly called. The push style for this operation is as follows:

```
print_item(Item passed_item)
    {
    passed_item.print()
    }
a.list.for_each(print_item)
```

The for_each() method iterates through a_list. For each item, for_each() calls the print_item() method, which is passed the current item on the list.

For each language, the actual code for the push style is different. For example, in C++, you can use the for_each() function in the Standard Template Library (STL). With this function, each item in the vector is pushed to the print_item() function.

```
void print_item(Item item)
  {
  cout << item <<' ';
  }
vector <Item> a_list;
for_each(a_list.begin(), a_list.end(), print_item);
```

In Ruby, the code could be as follows:

```
a_list = [1,2,3]
a_list.each { |passed_item| passeditem.print_item()}.
```

PUSH STYLE

Advantage—can be simpler code, once paradigm is understood

[3] *Design Patterns* refers to pull and push styles for a collection as *internal* and *external* iterators.

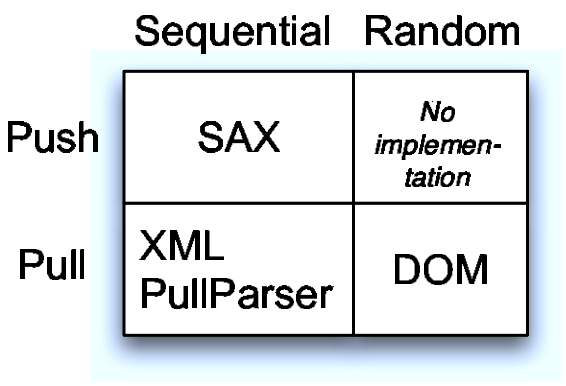

Figure 3.2: Examples of data interfaces

PULL STYLE
: Advantage—appears as a common control style (e.g., loop) in multiple languages

### One from Each Column

Pull/push style and sequential/random styles can be intermixed in combinations. As an example of a set of combinations in a specific area, let's revisit XML parsing. SAX is push/sequential; DOM is pull/random. There is also a pull-style sequential interface called XMLPullParser.[4]

Figure 3.2 shows how these three styles relate. The "No implementation" box shows a variation for which no current implementation exists.[5] Depending on what elements you want to retrieve from an XML file, what you want to do with the elements, and memory constraints, you choose one of these interface styles to create simpler code. To compare how you might employ each of these versions, let's take a look at some logic in pseudocode. In each of these examples, we print the count for one element in an XML file. The XML file looks like this:

[4] See http://www.xmlpull.org/v1/doc/api/org/xmlpull/v1/XmlPullParser.html for full details.

[5] We can't think of a need for this variation, so that may be why no one has created one.