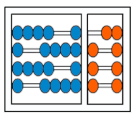
MC322 - Object Oriented Programming
Lesson 3
How to Think in Terms of Objects

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP



Structured and Object-Oriented Development

- Any fundamental change in thinking is not trivial.
- One side-effect of this debate is the misconception that structured and object-oriented development are mutually exclusive.
- As we know from wrappers, structured and object-oriented development coexist.
- In fact, when you write an OO application. I have never seen OO code that does not use loops, if-statements, and so on.
- Yet making the switch to OO design does require a different type of investment.

Three important things you can do to develop a good sense of the OO thought process are covered in this chapter:

- Knowing the difference between the interface and implementation
- Thinking more abstractly
- Giving the user the minimal interface possible

Knowing the Difference Between the Interface and the Implementation

Interface vs. Implementation

- One key aspect of strong object-oriented design is understanding the difference between the interface and the implementation.
- When designing a class, it's crucial to distinguish between what the user needs to know and what they don't.
- Encapsulation provides the mechanism for data hiding, ensuring that nonessential data is hidden from the user.
- In the toaster example from the last class, all appliances access the required electricity by complying with the correct interface.
- The toaster, for instance, doesn't need to know how the electricity is produced; it only needs to ensure compatibility with the interface.

Caution

Do not confuse the interface concept with the graphical user interface (GUI). As used here, the term interfaces is more general and is not restricted to a graphical interface.

Example: Interface and Implementation in an Automobile

- The interface between you and the car includes components such as the steering wheel, gas pedal, brake, and ignition switch.
- For most people, aesthetic issues aside, the main concern when driving a car is that the car starts, accelerates, stops, steers, and so on.
- The implementation, basically the stuff you don't see, is of little concern to the average driver.
- Most people would not even be able to identify certain components, but any driver would recognize and know how to use the steering wheel.

- If, however, a manufacturer decided to install a joystick instead of the steering wheel, most drivers would balk at this, and the automobile might not be a big seller.
- On the other hand, as long as the performance and aesthetics didn't change, the average driver would not notice if the manufacturer changed the engine (part of the implementation) of the automobile.

Considerations on Interchangeable Components

- It must be stressed that the interchangeable engines must be identical in every way—as far as the interface goes.
- Replacing a four-cylinder engine with an eight-cylinder engine would change the rules and likely would not work with other components that interface with the engine.
- Just as changing the current from AC to DC would affect the rules in the power plant example.

- The engine is part of the implementation, and the steering wheel is part of the interface.
- A change in the implementation should have no impact on the driver, whereas a change to the interface might.
- The driver would notice an aesthetic change to the steering wheel, even if it performs in a similar manner.
- It must be stressed that a change to the engine that is noticeable by the driver breaks this rule.

What Users See

- Interfaces also relate directly to classes.
- End users do not normally see any classes—they see the GUI or command line.
- However, programmers would see the class interfaces.
- Class reuse means that someone has already written a class.
- Thus, a programmer who uses a class must know how to get the class to work properly.

- This programmer will combine many classes to create a system.
- The programmer is the one who needs to understand the interfaces of a class.
- Therefore, when we talk about users in this chapter, we primarily mean designers and developers—not necessarily end users.
- Thus, when we talk about interfaces in this context, we are talking about class interfaces, not GUIs.

The Interface

- The services presented to an end user comprise the interface.
- In the best case, only the services the end user needs are presented.
- Of course, which services the user needs might be a matter of opinion.
- If you put 10 people in a room and ask each of them to do an independent design, you might receive 10 totally different designs—and there is nothing wrong with that.
- However, as a rule of thumb, the interface to a class should contain only what the user needs to know.
- In the toaster example, the user only needs to know that the toaster must be plugged into the interface (which in this case is the electrical outlet) and how to operate the toaster itself.

Identifying the User

Perhaps the most important consideration when designing a class is identifying the audience, or users, of the class.

The Implementation

- The implementation should be hidden from the user and A change to the implementation should not require a change to the user's code.
- The interface includes the syntax to call a method and return a value. If it not change, the user does not care whether the implementation is changed.
- We see this all the time when using a cell phone. To make a call, the interface is simple—we dial a number. Even if the provider changes equipment.
- Fundamental interface changes, like an area code change, do require the users to change behavior.

An Interface/Implementation Example

Let's create a simple (if not very functional) database reader class:

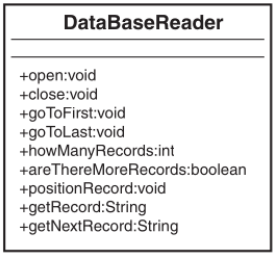
- We must be able to open and close a connection to the database.
- We must be able to position the cursor on the first and last record in the database.
- We must be able to find the number of records in the database.
- We must be able to determine whether there are more records in the database (that is, if we are at the end).
- We must be able to position the cursor at a specific record by supplying the key.
- We must be able to retrieve a record by supplying a key.
- We must be able to get the next record, based on the cursor's position.

Database query example

```
SELECT article, dealer, price
FROM shop
WHERE price=19.95
```

Database Reader Class Interface

- The interface is essentially the application-programming interface (API) that enables programmer to use a database.
- These methods are wrappers that enclose the functionality provided by the database system.
- Why would we do this? The short answer is that we might need to customize some database functionality. We may want to change the database engine itself without changing the code.
- Figure shows a class diagram representing a possible interface to the DataBaseReader class.



UML class diagram for the DataBaseReader class.

Public Interface

Remember that if a method is public, an application programmer can access it, and thus, it is considered part of the class interface. Do not confuse the term interface with the keyword interface used in Java and .NET—this term is discussed later.

Understanding Class Requirements

- To effectively use this class, do you, as a programmer, need to know anything else about it?
- Do you need to know how the internal database code actually opens the database?
- Do you need to know how the internal database code physically positions itself over a specific record?
- Do you need to know how the internal database code determines whether there are any more records left?

On all counts the answer is a resounding no! All you care about is that you get the proper return values and that the operations are performed correctly. The application programmer will most likely be at least one more abstract level away from the implementation.

Minimal Interface

Although perhaps extreme, one way to determine the minimalist interface is to initially provide the user no public interfaces. Of course, the class will be useless; however, this forces the user to come back to you and say, "Hey, I need this functionality." Then you can negotiate. Thus, you add interfaces only when it is requested. Never assume that the user needs something.

Creating Wrappers

Wrappers might seem overkill, but writing them has many advantages. Consider the problem of mapping objects to a relational database. OO databases are perfect for OO applications. However, most companies have years of data in legacy relational database systems. How can a company embrace OO technologies and stay on the cutting edge while retaining its data in a relational database?

Object Persistence

Object persistence refers to the concept of saving the state of an object so that it can be restored and used later. An object that does not persist dies when it goes out of scope. For example, the state of an object can be saved in a database.

Standalone Application

Even when creating a new OO application from scratch, avoiding legacy data might be difficult. This is because even a newly created OO application is most likely not a standalone application and might need to exchange information stored in relational databases (or any other data storage device, for that matter).

Code Example

What would the code for this public interface look like:

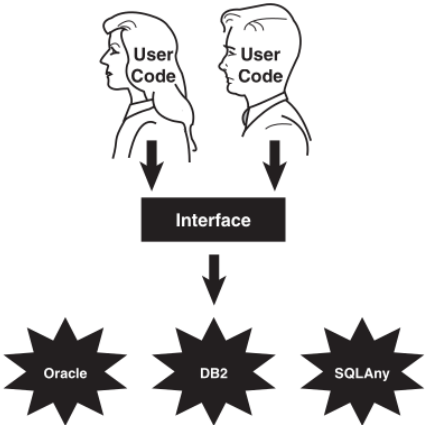
```
public void open(String Name){
    /* Some application-specific processing */
    /* call the Oracle API to open the database */
    /* Some more application-specific processing */
};

public void open(String Name){
    /* Some application-specific processing */
    /* call the SQLAnywhere API to open the database */
    /* Some more application-specific processing */
};
```

- To our great chagrin, this morning not one user complained.
- This is because the interface did not change even though the implementation changed! As far as the user is concerned, the calls are still the same.
- The code change for the implementation might have required quite a bit of work (and the module with the one-line code change would have to be rebuilt), but not one line of application code that uses this DataBaseReader class needed to change.

Separating Interface from Implementation

By separating the user interface from the implementation, we can save a lot of headaches down the road. In Figure, the database implementations are transparent to the end users, who see only the interface.



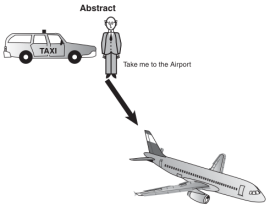
Using Abstract Thinking When Designing Interfaces

Advantages of Object-Oriented Programming

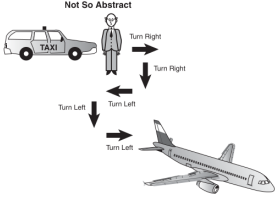
- One of the main advantages of OO programming is that classes can be reused.
- In general, reusable classes tend to have more abstract interfaces than concrete. Concrete interfaces tend to be very specific, whereas abstract interfaces are more general.
- However, simply stating that a highly abstract interface is more useful than a highly concrete interface, although often true, is not always the case.
- It is possible to write a beneficial, concrete class that is not reusable. This happens constantly, and there is nothing wrong with it in some situations.
- However, we are now in the design business and want to take advantage of what OO offers us. So our goal is to design abstract, highly reusable classes—and to do this, we will design highly abstract user interfaces.

Abstract vs. Concrete Interface

- To illustrate the difference between an abstract and a concrete interface, let's create a taxi object.
- It is much more useful to have an interface such as "drive me to the airport" than to have separate interfaces such as "turn right," "turn left," "start," "stop," and so on because all the user wants to do is get to the airport.
- When you emerge from your hotel, throw your bags into the back seat of the taxi and get in. The cabbie will turn to you and ask, "Where do you want to go?" You reply, "Please take me to airport X."
- You might not even know how to get to the airport, and even if you did, you wouldn't want to have to tell the cabbie when to turn and which direction to turn.
- How the cabbie implements the actual drive is of no concern to you, the passenger.



An abstract interface.



Not so abstract interface.

Abstract Interface and Reusability

- Now, where does the connection between abstract and reuse come in? Ask yourself which of these two scenarios is more reusable, the abstract or the not-so-abstract?
- To put it more simply, which phrase is more reusable:"Take me to the airport" or "Turn right, then right, then left, then left, then left"? Obviously, the first phrase is more reusable.
- You can use it in any city, whenever you get into a taxi and want to go to the airport.
- The second phrase will only work in a specific case.
- Thus, the abstract interface "Take me to the airport" is generally the way to go for a good, reusable OO design whose implementation would be different in Chicago, New York, or Cleveland.

Giving the User the Minimal Interface Possible

Designing Classes: Rules to Follow

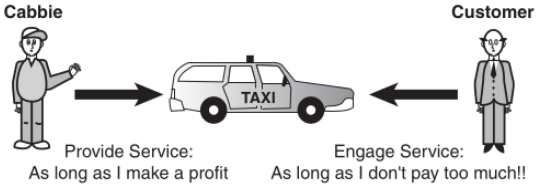
- Give the users only what they need. In effect, this means the class has as few interfaces as possible.
  - When designing a class, start with a minimal interface. It is better to have to add interfaces because users really need it than to give the users more interfaces than they need.
  - There are times when it is problematic for the user to have certain interfaces. For example, you don't want an interface that provides salary information to all users—only the ones who need to know.
  - Imagine handing a user a PC box without a monitor or a keyboard. Obviously, the PC would be of little use. You have just provided the user with the minimal set of interfaces to the PC. Of course, this minimal set is insufficient, and it immediately becomes necessary to add interfaces.

Designing Classes: Rules to Follow

- Public interfaces define what the users can access. If you initially hide the entire class from the user by making the interfaces private, when programmers start using the class, you will be forced to make certain methods public—these methods thus become the public interface.
- It is vital to design classes from a user's perspective, not an information systems viewpoint. Too often designers of classes (not to mention any other kind of software) design the class to make it fit into a specific technological model.
- Even if the designer takes a user's perspective, it is still probably a technician user's perspective, and the class is designed with an eye on getting it to work from a technology standpoint and not from ease of use for the user.
- Make sure when designing a class that you review the requirements and the design with the people who will use it—not just developers. The class will most likely evolve and must be updated when a system prototype is built.

Identifying Users: Taxi Example

- The taxi system users include the customers and the cab drivers.
- While customers are direct users, cab drivers must also be able to provide the service to the customers successfully.
- Providing interfaces that please customers but don't align with the capabilities or goals of cab drivers will not work.
- In software development, users may request certain functions, but if they are technically impossible to implement, they cannot be satisfied.
- Any object that sends a message to the taxi object is considered a user, including both customers and cab drivers.



Looking Ahead

The cabbie is most likely an object as well.

Object Behavior

- Identifying the users is only a part of the exercise.
- After the users are identified, you must determine the behaviors of the objects.
- From the viewpoint of all the users, begin identifying the purpose of each object and what it must do to perform properly.
- Note that many of the initial choices will not survive the final cut of the public interface.
- These choices are identified by gathering requirements using various methods such as UML UseCases.

Environmental Constraints

- In their book Object-Oriented Design in Java, Gilbert and McCarty point out that the environment often imposes limitations on what an object can do.
- In fact, environmental constraints are almost always a factor.
- Computer hardware might limit software functionality.
- For example, a system might not be connected to a network, or a company might use a specific type of printer.
- In the taxi example, the cab cannot drive on a road if a bridge is out, even if it provides a quicker way to the airport.

Identifying the Public Interfaces

With all the information gathered about the users, the object behaviors, and the environment, you need to determine the public interfaces for each user object.

So think about how you would use the taxi object:

- Get into the taxi.
- Tell the cabbie where you want to go.
- Pay the cabbie.
- Give the cabbie a tip.
- Get out of the taxi.

What do you need to do to use the taxi object?

- Have a place to go.
- Hail a taxi.
- Pay the cabbie money.

Initially, you think about how the object is used and not how it is built.You might discover that the object needs more interfaces, such as “Put luggage in the trunk” or “Enter into a mindless conversation with the cabbie.”

Identifying the Public Interfaces

- As is always the case, nailing down the final interface is an iterative process.
- For each interface, you must determine whether the interface contributes to the operation of the object. If it does not, perhaps it is not necessary.
- Many OO texts recommend that each interface model only one behavior.
- This returns us to the question of how abstract we want to get with the design. If we have an interface called enterTaxi(), we certainly do not want enterTaxi() to have logic in it to pay the cabbie. If we do this, then not only is the design somewhat illogical, but there is virtually no way that a user of the class can tell what has to be done to simply pay the cabbie.

Identifying the Implementation

- After the public interfaces are chosen, you need to identify the implementation.
- After the class is designed and all the methods required to operate the class properly are in place, the specifics of how to get the class to work are considered.
- Technically, anything that is not a public interface can be considered the implementation. This means that the user will never see any of the methods that are considered part of the implementation, including the method's signature (which includes the name of the method and the parameter list), as well as the actual code.
- It is possible to have a private method that is used internally by the class. Any private method is considered part of the implementation given that the user will never see it and thus will not have access to it. For example, a class may have a changePassword() method; however, the same class may have a private method that encrypts the password.
- This means that, theoretically, anything that is considered the implementation might change without affecting how the user interfaces with the class.

Conclusion

- Explore three areas for starting on the path to thinking in an object-oriented (OO) way.
- Note the absence of a definitive list of issues related to the OO thought process.
- Emphasize that practicing OO thinking is more of an art than a science.
- Encourage readers to develop their own ways of describing OO thinking.
- Preview Chapter 3, "Advanced Object-Oriented Concepts," which will discuss the life cycle of objects.
- Highlight that objects go through various states during their life cycle, with examples like the DataBaseReader object transitioning based on database status.

This lesson is based on Chapter 2 of "The Object-Oriented Thought Process".

MC322 - Object Oriented Programming
Lesson 3
How to Think in Terms of Objects

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP