MC322 - Object Oriented Programming
Lesson 5.1
Class Design Guidelines

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP

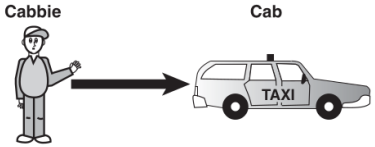
Designing Solid Classes

- OO programming supports the idea of creating classes that are complete packages, encapsulating the data and behavior of a single entity.
- A class should represent a logical component, such as a taxicab.
- This chapter presents several suggestions for designing solid classes.
- Obviously, no list such as this can be considered complete.
- You will undoubtedly add many guidelines to your personal list as well as incorporate useful guidelines from other developers.

Modeling Real World Systems

Object-Oriented Programming (OO)

- OO programming aims to model real-world systems like human thought processes.
- Designing classes are fundamental in OO programming, encapsulating data and behavior into interacting objects.
- Unlike structured approaches, where data and behavior are separate, OO encapsulates them into interacting objects.
- Classes model real-world objects and their interactions, mirroring human interactions.
- When creating classes, they should represent the true behavior of the object.
- Mistakes in transitioning to OO include creating classes with behavior but no class data, resembling structured models.
- Such an approach misses the power of encapsulation, failing to take advantage of the object-oriented paradigm.



A cabbie and a cab are real-world objects.

Identifying the Public Interfaces

Designing Class Interfaces

- The primary concern in class design is to minimize the public interface.
- The purpose of a class is to offer something useful and concise to the client.
- According to Gilbert and McCarty in "Object-Oriented Design in Java," the interface of a well-designed object describes the services that the client wants accomplished.
- If a class does not provide a useful service to a user, it should not have been built.

Extending the Interface

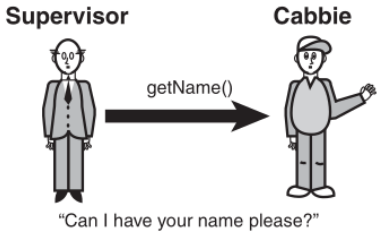
Even if the public interface of a class is insufficient for a certain application, object technology easily allows the capability to extend and adapt this interface by means of inheritance. In short, if designed with inheritance in mind, a new class can inherit from an existing class and create a new class with an extended interface.

Minimum Public Interface

- Providing the minimum public interface makes the class concise.
- The goal is to offer the user the exact interface needed to perform the task effectively.
- Incomplete public interfaces lead to users being unable to accomplish the full task.
- Improperly restricted public interfaces can result in unnecessary or dangerous access to behavior, leading to debugging and potential system integrity and security issues.
- Class creation involves a business proposition.
- Users should be involved in the design process from the start to ensure utility and proper interfaces.

Illustration: Cabbie Example

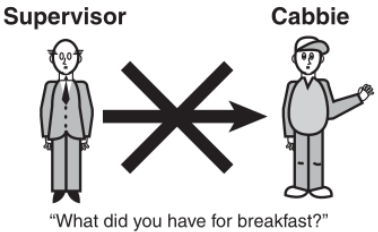
- Consider the cabbie example once again.
- If other objects in the system need to get the name of a cabbie, the Cabbie class must provide a public interface to return its name, such as the `getName()` method.
- For instance, if a Supervisor object needs a name from a Cabbie object, it must invoke the `getName()` method from the Cabbie object. In essence, the supervisor is requesting the cabbie for its name.
- Users of your code need to know nothing about its internal workings. All they need to know is how to instantiate and use the object. Provide them a way to get there but hide the details.



The public interface specifies how the objects interact.

Hiding the Implementation

- Hiding the implementation is essential.
- Identifying the public interface revolves around the users of the class. The implementation should not involve the users.
- A class is most useful if the implementation can change without affecting the users' code.
- For example, the Cabbie class might contain behavior related to how it eats breakfast. This behavior is part of the implementation of the Cabbie object and should not be available to other objects in the system.
- According to Gilbert and McCarty, the prime directive of encapsulation is that "all fields shall be private," ensuring that none of the fields in a class is accessible from other objects.



Objects don't need to know some implementation details.

Designing Robust Constructors (and Perhaps Destructors)

Constructors and Destructors

- A constructor should first and foremost put an object into an initial, safe state. This involves attribute initialization and memory management.
- It's important to ensure the object is constructed properly in the default condition. Providing a constructor to handle this default situation is usually a good idea.
- In languages with destructors, it's vital that destructors include proper clean-up functions. This often involves releasing system memory that the object acquired.
- Java and .NET reclaim memory automatically via garbage collection. However, in languages like C++, the developer must include code in the destructor to properly free memory.
- Ignoring this function can result in a memory leak.

Memory leaks

Memory leaks occur when objects fail to release the memory they acquired during their lifecycle. If objects are created and destroyed repeatedly without releasing memory, the available system memory slowly depletes. Eventually, the system may run out of memory, causing applications to become unstable or even lock up the system.

Designing Error Handling into a Class

Designing Error Handling into a Class

- Designing how a class handles errors is of vital importance, similar to the design of constructors.
- Error handling is discussed in detail in Chapter 3.
- Every system will encounter unforeseen problems, so ignoring potential errors is not a good idea.
- A developer of a good class anticipates potential errors and includes code to handle these conditions when they are encountered.
- The application should never crash. When an error is encountered, the system should either fix itself and continue or exit gracefully without losing any important user data.

Designing with Reuse in Mind

Documenting a Class and Using Comments

- Comments and documentation are essential aspects of good design.
- Unfortunately, they are often not taken seriously or ignored.
- While most developers know they should document their code, they often don't want to invest the time.
- However, good documentation practices are crucial for a good design.
- At the class level, developers might get away with inadequate documentation, but when the class is passed to others for extension or maintenance, or becomes part of a larger system, proper documentation becomes vital.
- Lack of documentation and comments can be detrimental in these scenarios.

Too Much Documentation

Overcommenting can be a problem, as excessive documentation or commenting can become noise and defeat the purpose. Unfocused documentation is often ignored.

Building Objects with the Intent to Cooperate

- In Chapter 6, "Designing with Objects," we discuss many design issues involved in building a system (might not be covered in class, please read it).
- Almost no class lives in isolation; typically, there's no reason to build a class if it won't interact with others.
- Interactions between classes are a fundamental aspect of class life.
- Classes will either service other classes, request services from them, or both.
- In later chapters, various ways of class interactions are discussed.
- In the cabbie example, both the cabbie and the supervisor are not standalone entities; they interact with each other at various levels.
- When designing a class, it's crucial to consider how other objects will interact with it.

Designing for Reuse

- Objects can be reused in different systems, and code should be written with reuse in mind.
- For instance, once a Cabbie class is developed and tested, it can be used wherever a cabbie is needed.
- To ensure a class can be reused in various systems, it must be designed with reuse in mind.
- Designing for reuse requires careful thought during the design process.
- It's not a trivial task to anticipate all the possible scenarios in which an object must operate.

Designing with Extensibility in Mind

Extending Classes with Inheritance

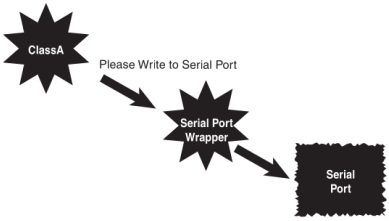
- Adding new features to a class often involves extending an existing class, adding new methods, and modifying existing behavior.
- Inheritance plays a crucial role here, allowing new classes to inherit attributes and behaviors from existing ones.
- For example, if you have a Person class, you might later want to create an Employee class or a Vendor class. In this case, having Employee inherit from Person can be a good strategy, making Person extensible.
- Design classes to be extensible, so they can be easily subclassed without restrictions on future functionalities.
- **Abstraction guideline**: classes should contain only data and behaviors specific to their purpose, allowing other classes to subclass and inherit appropriate data and behaviors.

Attributes and Methods as Static

It's crucial to determine which attributes and methods can be declared as static. Static attributes and methods are shared by all objects of a class.

Abstracting Out Nonportable Code

- If your system requires nonportable code (i.e., code that runs only on specific hardware platforms), it's crucial to abstract this code out of the class.
- Abstraction involves isolating the nonportable code in its own class or method, which can be overridden if needed.
- For instance, when dealing with code that accesses hardware-specific features like a serial port, create a wrapper class to handle it.
- Your primary class should then interact with this wrapper class to access the necessary information or services, rather than embedding system-dependent code directly.
- If the class moves to another hardware system, the way to access the serial port changes, or you want to go to a parallel port, the code in your primary class does not have to change.



A serial port wrapper.

Keeping the Scope as Small as Possible

- Keeping the scope small is essential for abstraction and hiding the implementation.
- The goal is to localize attributes and behaviors as much as possible, making maintenance, testing, and extension easier.
- For instance, if a method requires a temporary attribute, it's best to keep it local.
- Consider the example where the attribute 'temp' is unnecessarily in the class level:

```
public class Math {
    int temp=0;
    public int swap (int a, int b) {
        temp = a;
        a=b;
        b=temp;
        return temp;
    }
}
```

- In this case, 'temp' is only needed within the 'swap()' method, so it should be moved within its scope:

```
public class Math {
    public int swap (int a, int b) {
        int temp=0;
        temp = a;
        a=b;
        b=temp;
        return temp;
    }
}
```

- Keeping the scope as small as possible simplifies code and improves its maintainability.

A Class Should Be Responsible for Itself

- In their book "Java Primer Plus," Tyma, Torok, and Downing propose the guideline that all objects should be responsible for acting on themselves whenever possible.
- Consider an example of printing a circle. In a non-OO approach, you might have:

```
print(circle);
```

- Functions like print, draw, etc., would require a case statement or if/else structure to determine how to handle each shape passed to them.
- For instance, separate print routines for each shape could be called:

```
printCircle(circle);
printSquare(square);
```

- However, following the object-oriented principle, each class should handle its own operations whenever feasible, simplifying the code and improving maintainability.

A Class Should Be Responsible for Itself

- When adding a new shape, all functions handling shapes need to be updated to accommodate the new shape.
- For instance, a switch statement might need to be updated:

```
switch (shape) {
    case 1: printCircle(circle); break;
    case 2: printSquare(square); break;
    case 3: printTriangle(triangle); break;
    default: System.out.println("Invalid
        shape."); break;
}
```

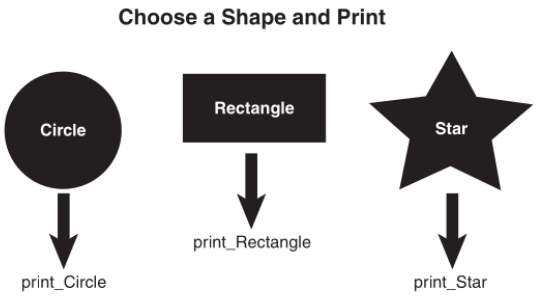
- This approach leads to code duplication and increases the complexity of maintenance.

- In an object-oriented example, polymorphism is used to group shapes like Circle into a Shape category.
- With polymorphism, Shape can figure out what type of shape it is and know how to print itself.
- For example:

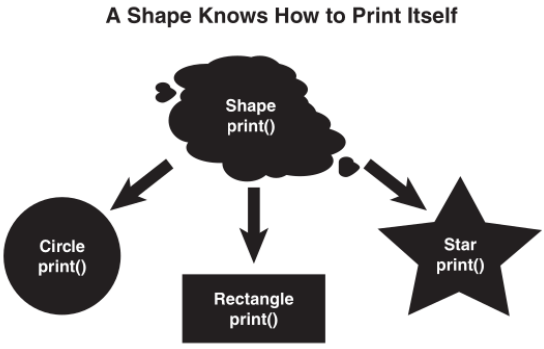
```
Shape.print(); // Shape is actually a Circle
Shape.print(); // Shape is actually a Square
```

- The key point is that the call is identical; the context of the shape dictates how the system reacts.

A Class Should Be Responsible for Itself



A non-OO example of a print scenario.



An OO example of a print scenario.

Designing with Maintainability in Mind

Designing for Maintainability

- Designing useful and concise classes promotes a high level of maintainability.
- Just as you design a class with extensibility in mind, you should also design with future maintenance in mind.
- The process of designing classes forces you to organize your code into manageable pieces.
- Separate pieces of code are typically more maintainable than larger ones.
- A key strategy to promote maintainability is to reduce interdependent code. Changes in one class should ideally have minimal or no impact on other classes.

Highly Coupled Classes

Classes that are highly dependent on one another are considered highly coupled. When a change in one class necessitates a change in another, these classes are deemed highly coupled. Conversely, classes with no such dependencies exhibit a low degree of coupling. For further insights on this topic, refer to "The Object Primer" by Scott Ambler.

Maintaining Low Coupling

- Properly designed classes should require changes only to the implementation, avoiding modifications to the public interface whenever possible.
- Changes to the public interface can lead to ripple effects throughout all systems using the interface.
- For instance, altering the getName() method of the Cabbie class would necessitate changes in every system utilizing this interface, which can be a daunting task.
- To ensure a high level of maintainability, strive to keep the coupling level of your classes as low as possible.

Using Iteration

- In both design and programming functions, employing an iterative process is highly recommended.
- This approach aligns well with the concept of providing minimal interfaces.
- A robust testing plan helps identify areas where interfaces may be insufficient, allowing the process to iterate until the class has the appropriate interfaces.
- Testing isn't limited to coding; conducting design walkthroughs and other review techniques is also valuable.
- Testers benefit from iterative processes as they are involved early in the process, rather than receiving a system hastily at the end of development.

Testing the Interface

- Minimal implementations of interfaces are often referred to as stubs.
- Stubs allow you to test interfaces without writing actual code.
- Instead of connecting to a real database, stubs can be used to verify that interfaces are functioning correctly from the user's perspective.
- At this stage, the implementation is not necessary as the design of the interface may still be evolving, and completing the implementation prematurely could waste time and energy.
- When a user class interacts with the DataBaseReader class, the information returned is provided by code stubs instead of the actual database, which may not yet exist.
- Once the interface is complete and implementation begins, the database can be connected, and the stubs can be replaced.

Code Example: Simulated Database

```
public class DataBaseReader {
    private String db[] = { "Record1",
                            "Record2",
                            "Record3",
                            "Record4",
                            "Record5" };
    private boolean DBOpen = false;
    private int pos;

    public void open(String Name){
        DBOpen = true;
    }

    public void close(){
        DBOpen = false;
    }

    public void goToFirst(){
        pos = 0;
    }

    public void goToLast(){
        pos = 4;
    }
```

```
    public int howManyRecords(){
        int numOfRecords = 5;
        return numOfRecords;
    }

    public String getRecord(int key){
        /* DB Specific Implementation */
        return db[key];
    }

    public String getNextRecord(){
        /* DB Specific Implementation */
        return db[pos++];
    }
}
```

Simulating Database Calls

- The methods in the code example simulate various database operations.
- The strings within the array represent the records that would be retrieved from a real database.
- For instance, methods like ‘getRecord()‘ and ‘getNextRecord()‘ return the records stored in the array, mimicking database queries.
- Once the actual database is integrated into the system, the array will be replaced with real database calls.
- As you find problems with the interface design, make changes and repeat the process until you are satisfied with the result.

Keeping the Stubs Around

When you’re finished with stubs, avoid deleting them. Instead, keep them in the code for potential future use, ensuring that users cannot access them. Well-designed programs often integrate test stubs into the design and retain them in the program for later use. In essence, design testing directly into the class!

Using Object Persistence

Object Persistence

- Object persistence is a crucial concern in many object-oriented (OO) systems.
- Persistence involves maintaining the state of an object over time.
- When a program is run, if the object's state is not saved in some manner, the object ceases to exist and cannot be recovered.
- While transient objects may suffice for some applications, in most business systems, it's essential to save the state of objects for later use.

Object Persistence

Object persistence, along with the topics in the next section, may not be traditional design guidelines, but they are crucial considerations when designing classes. Introducing them early emphasizes their importance and underscores the need to address them during the class design phase.

Object Persistence Mechanisms

- In its simplest form, object persistence involves serializing an object and writing it to a flat file.
- Modern technology favors XML-based persistence methods.
- While an object can theoretically persist in memory as long as it's not destroyed, we'll focus on storing persistent objects on storage devices.
- There are three primary storage devices to consider:
  - Flat file system: Objects can be stored in a flat file by serialization, although this has limited use.
  - Relational database: Middleware is typically required to convert objects to a relational model for storage.
  - Object-oriented (OO) database: This is the ideal method for persisting objects, but many companies still rely on legacy systems, necessitating interface with legacy data.

Serializing and Marshaling Objects

- Using objects in environments originally designed for structured programming poses challenges.
- Middleware examples, like writing objects to relational databases, highlight this issue.
- Problems also arise when attempting to write an object to a flat file or send it over a network.
- To send an object over a wire (e.g., to a file or over a network), the system must deconstruct the object (flatten it out), transmit it, and then reconstruct it on the receiving end. This process is known as serializing an object.
- The act of sending the serialized object across a wire is called marshaling.
- A serialized object can theoretically be written to a flat file and later retrieved in the same state in which it was written.
- A key concern is ensuring that the serialization and deserialization processes use the same specifications, much like an encryption algorithm. Java provides the `Serializable` interface to facilitate this translation.

Conclusion

Conclusion

- This chapter presents numerous guidelines for designing classes, although it's not an exhaustive list.
- Additional guidelines will likely be encountered as you delve deeper into object-oriented (OO) design.
- While this chapter focuses on design issues concerning individual classes, it's important to recognize that classes do not exist in isolation.
- Classes must be designed to interact with other classes, forming systems that ultimately deliver value to end users.
- Chapter 6, "Designing with Objects," delves into the topic of designing complete systems, providing further insights into OO design.

MC322 - Object Oriented Programming
Lesson 5.1
Class Design Guidelines

Prof. Marcos M. Raimundo
Instituto de Computação - UNICAMP