Assignment Report for Assignment 02

|  |  |  |
| --- | --- | --- |
| Course and Section | CSC 340.01 |  |
| Assignment Name | Assignment 02 |
| Due Date and Time | 06-23-2023 @ 11:59 PM |
|  |  |
| First Name and Last Name | Nhan Nguyen |
| SFSU Email Account | nnguyen14@sfsu.edu |
| First Name and Last Name of Teammate | N/A |
| SFSU Email Account of Teammate | N/A |

**PART A**

**Question Description and Analysis**:

This part of the assignment asks me to choose 5 guidelines in [*Y. Daniel Liang’s 8 Class Design Guidelines*](http://csc340.ducta.net/WEEK-01/JAVAtoCPP-ClassDesignGuidelines.pdf) and discuss them in depth, using at least one page and maybe some codes for each guideline to demonstrate my points.

**Answer**:

1. **Cohesion**

In Java programming, maintaining cohesion is crucial for writing clean, maintainable, and efficient code. Cohesion refers to the extent to which the elements of a class belong together and work towards a single purpose. By adhering to the guideline of ensuring cohesion, developers can create classes that accurately represent individual entities and separate responsibilities effectively. A key aspect of cohesion is ensuring that a class represents a single entity. This means that all the properties, methods, and operations within the class should be directly related to the purpose of that entity. By doing so, we create classes that are focused and encapsulate the necessary functionality. Let's consider an example where we need to model a Car class:

public class Car {

private String make;

private String model;

private int year;

// Constructor, getters, and setters

public void startEngine() {

// Code to start the car's engine

}

public void accelerate() {

// Code to accelerate the car

}

// Other car-related methods

}

In this example, all the properties (make, model, year) and methods (startEngine, accelerate) directly relate to the car entity. This ensures that the class represents a single entity, making it easier for developers to understand its purpose and functionality. By grouping all the relevant properties and methods together, the code becomes more readable, reducing cognitive load and making it simpler to reason about. This leads to enhanced comprehension and facilitates collaboration among team members. Moreover, when a class becomes responsible for too many tasks, it can become complex and difficult to manage. To address this, responsibilities can be separated by creating multiple classes, each focusing on a specific aspect or behavior. This modular design allows developers to isolate and understand individual components independently, simplifying code comprehension and reducing dependencies. Modularity also facilitates code reuse, as classes designed with clear responsibilities can be easily incorporated into other projects without carrying unnecessary baggage.

1. **Consistency**

Consistency in naming is crucial for creating code that is easy to understand and navigate. Developers should choose informative and meaningful names for classes, data fields, and methods. It is essential to avoid using different names for similar operations or variables that serve the same purpose. Consistency fosters clarity and enables developers to quickly identify the purpose and functionality of various components within the codebase.

In Java programming, a common convention is to place data field declarations before constructors and constructors before methods. This practice enhances code readability by providing a logical flow to the code structure. By declaring data fields at the beginning, developers can easily identify the properties associated with a class. Placing constructors next allows for clarity in defining how instances of the class are initialized. Finally, positioning methods after constructors aids in locating and understanding the behavior and functionality provided by the class.

Moreover, it is generally recommended to provide a public no-arg constructor for constructing a default instance of a class. This practice allows users to create objects without specifying any arguments, making the class more accessible and easier to use. If a class does not support a no-arg constructor, it is essential to document the reason for its absence. This documentation provides clarity to other developers who might use the class, ensuring they understand the intended usage and any requirements for instantiation.

In certain scenarios, preventing users from creating objects for a class may be desired. This can be achieved by declaring a private constructor in the class. The Math class in Java is a good example of this approach. By making the constructor private, it is explicitly communicated that the class is not meant to be instantiated, as it provides only static utility methods. Private constructors are a useful tool to enforce design patterns, limit object creation, or ensure that the class is used solely for static operations.

1. **Encapsulation**

Encapsulation is a fundamental principle in object-oriented programming that promotes data hiding and information hiding. In Java programming, encapsulation involves using the private modifier to hide class data from direct access by clients. Encapsulation ensures that the internal state and implementation details of a class are hidden from external clients. By declaring class data as private, we restrict direct access and modification from outside the class, preventing unintended modifications or invalid states. This preserves the integrity of the class, minimizing the risk of unwanted side effects and improving code stability.

Moreover, encapsulation allows us to control how data is accessed and modified by providing controlled access points. This is achieved through the use of getter and setter methods. Getter and setter methods play a crucial role in encapsulation by providing an interface for accessing and modifying private data fields. They act as intermediaries between the internal state of a class and external clients, allowing controlled and consistent interactions. Getter methods expose the values of private data fields, enabling clients to retrieve information while keeping the underlying data protected. Setter methods, when used appropriately, facilitate the controlled modification of data fields, providing a means to update the internal state of the class while enforcing any necessary validation or logic. By selectively providing getter and setter methods, developers can enforce read-only or write-only access to data fields as needed. This allows for fine-grained control over the accessibility and mutability of class data, ensuring data consistency and protecting sensitive information.

1. **Completeness**

Completeness in class design is a critical aspect emphasized by Y. Daniel Liang's guidelines, as it ensures that all necessary components are included to create well-structured and effective Java classes. Let's explore the concept of completeness further through example.

Consider a class named "Person" that represents individuals. To achieve completeness, this class should include appropriate data fields, constructors, and methods. For instance, the data fields could include attributes such as "name," "age," and "address." By encapsulating these fields and making them private, we ensure that they are not directly accessible from outside the class. Getter and setter methods can then be provided to enable controlled access to these fields.

public class Person {

private String name;

private int age;

private String address;

public Person(String name, int age, String address) {

this.name = name;

this.age = age;

this.address = address;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

public String getAddress() {

return address;

}

public void setAddress(String address) {

this.address = address;

}

}

In the example above, the Person class exhibits completeness by including relevant data fields and corresponding getter and setter methods. The constructors allow for different initialization scenarios, such as providing a name, age, and address, enabling flexibility when creating Person objects. In conclusion, completeness in class design, as exemplified by the Person classes, ensures the inclusion of all necessary components in a well-structured manner. By incorporating appropriate data fields, constructors, methods, and considering inheritance and interface implementation, developers can create classes that are readable, reusable, and adaptable. Adhering to these principles fosters code maintainability and facilitates efficient software development.

1. **Interfaces vs. Abstract Classes**

Interfaces provide a powerful mechanism for defining contracts and establishing common behavior across unrelated classes. They consist of abstract methods, which are methods without implementations, and constant variables. By implementing an interface, a class guarantees that it will provide concrete implementations for all the methods defined in the interface. Interfaces support multiple inheritance, allowing a class to implement multiple interfaces simultaneously. This enables greater flexibility and the ability to leverage polymorphism, as objects can be treated as instances of their interface types. Interfaces are commonly used to create abstractions, define standardized contracts, and promote loose coupling between components.

On the other hand, abstract classes serve as partial implementations for subclasses to inherit and extend. They are classes that cannot be instantiated on their own and can contain both abstract and concrete methods. Abstract classes allow developers to define common attributes and behaviors shared by a group of related classes. By providing default implementations for certain methods, abstract classes offer a foundation upon which subclasses can build and specialize. Abstract classes promote code reuse and provide a natural way to organize and structure related classes within an inheritance hierarchy. They are particularly useful when there is a need for shared functionality among a group of classes but the exact implementation may vary.

In summary, interfaces and abstract classes are essential tools in Java programming, offering distinct advantages based on their respective features. Interfaces enable the creation of contracts and support polymorphism, while abstract classes facilitate code reuse and provide a foundation for related classes. By understanding the characteristics and use cases of interfaces and abstract classes, developers can design more flexible, modular, and maintainable software systems.

**Screenshots of Outputs and Explanation**:

These screenshots show what I accomplished…

**PART B**

**Question Description and Analysis**:

This part of the assignment asks that…

**Answer**:

This is my answer…

**Screenshots of Outputs and Explanation**:

These screenshots show what I accomplished…