SAN FRANCISCO STATE UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

OBJECT-ORIENTED PROGRAMMING
CLASS DESIGN

Duc Ta

Class Design Guidelines

Y. Daniel Liang

1. Cohesion

- [✓] A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose.
- [✓] A single entity with many responsibilities can be broken into several classes to separate the responsibilities.

2. Consistency

- [✓] Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. A popular style is to place the data declaration before the constructor and place constructors before methods.
- [✓] Make the names consistent. It is not a good practice to choose different names for similar operations.
- [✓] In general, you should consistently provide a public no-arg constructor for constructing a default instance. If a class does not support a no-arg constructor, document the reason. If no constructors are defined explicitly, a public default no-arg constructor with an empty body is assumed.
- [✓] If you want to prevent users from creating an object for a class, you can declare a private constructor in the class, as is the case for the Math class.

3. Encapsulation

- [√] A class should use the *private* modifier to hide its data from direct access by clients. This makes the class easy to maintain.
- [√] Provide a getter method only if you want the data field to be readable, and provide a setter method only if you want the data field to be updateable.

4. Clarity

- [√] Cohesion, consistency, and encapsulation are good guidelines for achieving design clarity. Additionally, a class should have a clear contract that is easy to explain and easy to understand.
- [√] Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on how or when the user can use it, design the properties in a way that lets the user set them in any order and with any combination of values, and design methods that function independently of their order of occurrence.
- [√] Methods should be defined intuitively without causing confusion.
- [√] You should not declare a data field that can be derived from other data fields.

5. Completeness

- [✓] Classes are designed for use by many different customers. In order to be useful in a wide range of applications, a class should provide a variety of ways for customization through properties and methods.

6. Instance vs. Static

- [✓] A variable or method that is dependent on a specific instance of the class must be an instance variable or method. A variable that is shared by all the instances of a class should be declared static.
- [✓] Always reference static variables and methods from a class name (rather than a reference variable) to improve readability and avoid errors.
- [✓] Do not pass a parameter from a constructor to initialize a static data field. It is better to use a setter method to change the static data field.
- [✓] Instance and static are integral parts of object-oriented programming. A data field or method is either instance or static. Do not mistakenly overlook static data fields or methods. It is a common design error to define an instance method should have been static.
- [✓] A constructor is always instance, because it is used to create a specific instance. A static variable or method can be invoked from an instance method, but an instance variable or method cannot be invoked from a static method.

7. Inheritance vs. Aggregation

- **[✓]** The difference between inheritance and aggregation is the difference between an is-a and a has-a relationship.

8. Interfaces vs. Abstract Classes

- **[✓]** Both interfaces and abstract classes can be used to specify common behavior for objects. How do you decide whether to use an interface or a class? In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can modeled using interfaces.
- **[✓]** Interfaces are more flexible than abstract classes, because a subclass can extend only one superclass but can implement any number of interfaces. However, interfaces cannot contain concrete methods. The virtues of interfaces and abstract classes can be combined by creating an interface with an abstract class that implement it. Then you can use the interface or the abstract class, whichever is convenient.

See you next class!