OO Design Guidelines

Written by John Bell for CS 342, Spring 2018

Based on chapter 5 of “The Object-Oriented Thought Process” by Matt Weisfeld, and other sources.

Modeling Real-World Systems

• Respect the semantics of the real-world entities being modeled by each OO Class.
• Class objects should behave the same as their real-world counterparts.
• Do not add any data or functionality to a class that is not present in the real-world entity.
• If necessary, create a separate class to implement extended features, and apply delegation or inheritance.
• In design, consider which class(es) should be responsible for delivering which services. At a minimum, each class is responsible for its own data.
Identifying the Public Interfaces

• The public interface defines the class to the world.
• Strive for the minimum public interface.
• All data and some methods should be private or protected.
• Minimized getters and setters whenever possible. Apply delegation instead. (Instead of getting stuff from an object, ask the object to act on the stuff.)
• Name methods according to the services performed, not the internal data being accessed.
  • Don’t give away clues regarding the internal implementation through method names or parameters.

Basic Methods that Should Always be Considered, Usually Provided

• Constructors ( & Destructors )
  • No-argument constructors
  • Copy constructors in C++ or clone( ) in Java
  • ~Destructors in C++ or finalize( ) in Java

• Print methods
  • toString( ) in Java
  • Operator << in C++

• Comparators
  • equals( ) in Java, or operator == in C++
  • May also need !=, <, >, etc. if sorting

• Type conversions, may be a form of constructor.
Designing Robust Constructors (and Perhaps Destructors)

- Constructors are responsible for ensuring all objects start out in an initial safe state.
- All fields defined in a given class should be addressed in the constructor for that class.
  - Descendant classes may override ancestor constructors if necessary, after the ancestor constructors have run.
- In languages without garbage collection, destructors are often vital to avoid memory leaks.
- Even with garbage detection, certain resources must be closed cleanly. (E.g. open files. finalize() )

Designing Error Handling Into a Class

- All programs encounter errors of some kind.
- A well-written program will not crash, but will exit cleanly when necessary due to errors.
- Above all, programs must fail safely, without loss of data and without inflicting any damage to lives or property.
- Exceptions are generally the preferred means of handling errors in OOP.
Designing with Reuse in Mind

• A well-written class should be usable in any application in which the real-world entity being modeled plays a part. It should not be specific to just the current application.
• Expect objects to interact with other objects.
• Document well, including but not limited to comments and descriptive variable / method names.
  • Consider Javadoc, Doxygen, or similar tools.
• Expect someone else to use your code in the future.

Designing with Extensibility in Mind

• What attributes and methods can be static, and/or final?
• Abstract out non-portable code. Encapsulate any non-portability into the smallest unit possible, and make it interchangeable with alternate versions.
• Keep scope as small as possible.
  • For example, don’t introduce a class variable if it is only needed by a single method. Use a local variable instead.
• Avoid global variables if at all possible. (If your language even allows them.)
Designing with Maintainability in Mind

• Keep coupling between classes as low as possible.
• To the greatest extent possible all classes should be independent.
• Small compact classes with minimal focused functionality are easier to maintain than large complex classes, even though there be more of them.
• Fix the interface as much as possible.
  • Test for the necessary interface with early stubs.
  • Develop iteratively to identify interface problems early.

Object Persistence

• Persistent data is that which must be remembered when a program exits, and restored upon reentry.
• Persistent objects can be stored in three ways:
  • A flat file system stores data in ordinary text or binary files. Possible though not generally optimal for objects.
  • A relational database generally consists of tables:
    • One table for each class type.
    • One row for each individual object
    • One column for each field.
    • Unique keys used to cross-reference objects & relations.
  • OO databases are specialized, and rarely used.
Marshalling and Serializing

• Sending an object over a wire, e.g. network, is termed **marshalling** the object.

• In order to marshall an object, it must first be **serialized** - converted to a serial flow of data - and then **deserialized** at the other end.

• The **Serializable** Interface in Java defines a class with methods for serializing and deserializing the class objects built into the class. Other languages have similar concepts.

• Serialization can be useful for other purposes, such as file storage, and is often proprietary.

Fields/Methods Defined in a Parent Class Must Apply to All Descendants

• If any attributes or methods defined in an ancestor class do not apply to all descendants of the class, then consider:
  • Should the fields/methods be pulled down into the subclasses in which they apply?
  • Is this a true inheritance situation? Can the descendants say that they truly ARE special cases of their ancestor if they do not need all the defined fields or methods?