Object- Oriented Design with UML and Java

Part I: Fundamentals

University of Colorado 1999 - 2002

CSCI-4448 - Object-Oriented Programming and Design

These notes as free PDF files:

`https://www.leberknight.com/csci4448.html`

Copyright © David Leberknight

Version 2023

What is this course all about?

- Learn the Object-Oriented paradigm.
- Learn why some designs are better than others.
- Learn how to implement these designs in Java.
- Learn the Unified Modeling Language (UML).
- Learn some Design Patterns.
- Learn some of the latest-and-greatest commercial OO technology.
- Learn a process to make best use of this technology.
- Be prepared for further study and to work on real-world projects.

Software Engineering *in the wild*...

Programming Evolution

- First there was Machine Language with **10101010**…
- Assembly Language provided **symbols**.
- High-level languages were invented to provide **structure** to the graph of program statements.
- **Data structures** and **algorithms** are reusable program structures.
- **Object-orientation** is primarily based on the problem to be solved rather than on the machine. The graph of object **collaborations** is at a higher-level of **abstraction**.
- **Design Patterns** provide reusable object structures**.**
- **Components** are reusable software entities.
- **Frameworks** such as Java E.E. **Containers** reuse distributed-transaction managers, user-session managers, support for object-to-relational-database mapping, dependency injection, and more.

Why Object-Oriented?

- OO is ***easier to comprehend***, for humans.
- The implementation can be less complex.
- There's a small conceptual gap between analysis and implementation.
- A well-designed set of objects is resilient.
- It's easier to reuse an class than a function.
- The modeling process creates a common vocabulary and shared understanding between developers and users / clients.
- You don't need to be a programmer to understand a UML model.
- Other benefits to be discussed...

These benefits are not automatic.

Design is an art.

Example: David walks his dog, Leroy

- *Find the objects...*

Linguistics & Cognition

Nouns are the primary words that humans use. We qualify them with modifiers and attributes. Then we associate them with verbs. Furthermore, we make heavy use of abstractions & generalizations...

- Object-oriented design follows this pattern.
- Procedural / functional design does not.

Subject-verb-object sentences flow from object models:

  - People own pets.
  - David owns Leroy.
  - A computer game player has a strategy.

The Procedural Approach

- The system is organized around procedures.
- Procedures send data to each other.
- Procedures and data are clearly separated.
- The programmer focuses on data structures, algorithms and sequencing.
- Functions are hard to reuse.
- Expressive visual modeling techniques are lacking.
- Concepts must be transformed between analysis & implementation.
- This paradigm is essentially an abstraction of machine / assembly language.

The Object-Oriented Approach

- Begin by modeling the problem domain as objects.
- The implementation is organized around objects.
- Objects send messages to each other.
- Related data and behavior are tied together.
- Visual models are expressive and easy to comprehend.
- Powerful concepts:
  - Encapsulation, interfaces, abstraction, generalization, inheritance, delegation, responsibility-driven design, separation of concerns, polymorphism, design patterns, reusable components, service-oriented architecture, message-oriented middleware, . . .

Example: Temperature Conversion

- The Procedural / Functional approach:

```
float c = getTemperature(); // assume Celcius
float f = toFarenheitFromCelcius( c );
float k = toKelvinFromCelcius( c );
float x = toKelvinFromFarenheit( f );
float y = toFarenheitFromKelvin( k );
```

- The OO approach:

```
Temp temp = getTemperature();
float c = temp.toCelcius();
float f = temp.toFarenheit();
float k = temp.toKelvin();
```

Objects

- Represent ***things***.
- Have ***responsibilities***.
- Provide ***services***.
- Exhibit ***behavior***.
- Have ***interfaces***.
- Have ***identity***.
- Send ***messages*** to other objects.
- Should be self-consistent, ***coherent***, and complete.
- Should be ***loosely coupled*** with other objects.
- Should ***encapsulate*** their ***state*** and internal structures.
- Should not be complex or large.

Encapsulation

- Exposing only the ***public interface***.
- Hiding the “gears and levers.”
- Protects the object from outside interference.
- Protects other objects from details that might change.
- ***Information hiding*** promotes ***loose coupling***.
- Reduces complex interdependencies.
- Good fences make good neighbors.

Example: ***Your car’s gas pedal.***

- Push down – go faster.

Encapsulation Example

- Best practice: Objects speak to each other by method calls not by direct access to attributes.

```
class Person {
  public int age; // yuk
}

class BetterPerson {
  private int age; // dateOfBirth ?
  public int getAge() { return age; }
}
```

Access Control

Keywords that determine the degree of encapsulation:

public = Interface stuff

private = Can only be accessed by the class' own member functions (in C++, also by the class' ***friends)***.

protected = Private, except for subclasses (in Java, protected attributes and methods are also available to classes in the same ***package***).

- Rule of thumb: make everything as inaccessible as possible.
- Make things private unless there's a good reason not to.
- Encapsulation is good.

Classes

- Programmers write ***code*** to define classes.
- An object is an ***instance*** of a class.
- An object, once instantiated, cannot change its class.
- A class defines both the interface(s) and the implementation for a set of objects, which determines their behavior.
- ***Abstract*** classes cannot have instances.
- *C**oncrete*** classes can.
- Some OO languages (such as Smalltalk) support the concept of a ***meta-class*** which allows the programmer to define a class on-the-fly, and then instantiate it.
- Java has a class called **`Class`**.

Class Attributes and Behaviors

- Class attributes are **shared** by all the instances of the class (indicated by the keyword “**static**”).
- Public and static items are essentially **global**.

Examples:

- An Employee class may be responsible to keep track of all employees. It could have a method to calculate the number of employees who are fully vested in a stock option scheme, say.
- A LotteryTicket class may use a seed to generate random numbers; that seed is shared by all instances of the class.

Abstraction

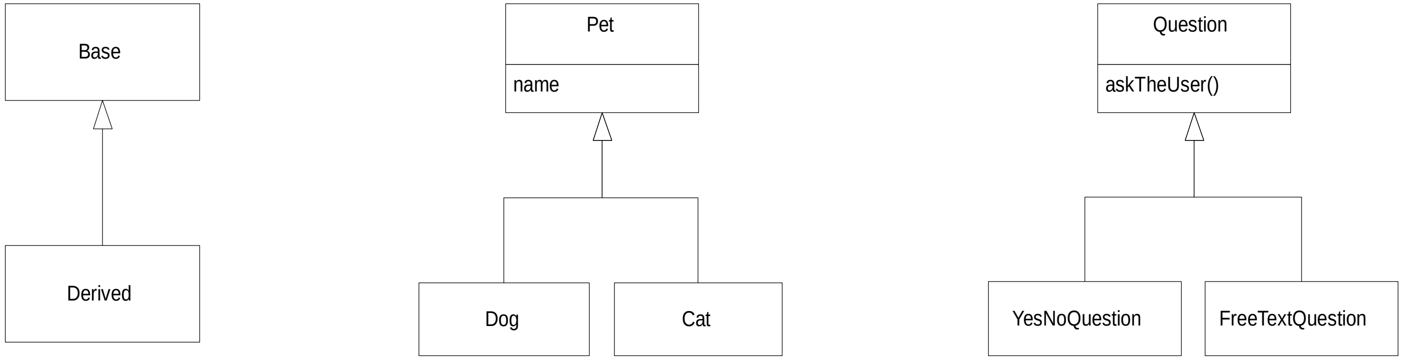
Abstraction allows ***generalizations***.

- Simplify reality - ignore complex details.
- Focus on commonalties but allow for variations.

Human beings often use generalizations.

- When you see a gray German Shepherd named Rex owned by Jane Doe... Do you think ***dog***?

Abstraction Vocabulary



- ***Base*** class = ***parent*** class = ***superclass***.
- ***Derived*** class = ***child*** class = ***subclass***.
- The Derived class ***inherits*** from the Base class; the Derived class ***extends*** the Base class; the Derived class is a ***specialization*** of the Base class.
- The Base class is a ***generalization*** of its Derived classes; one could say, "In general, all Pets have names."

Inheritance

- Implied by *is-a-kind-of* relationships.
  - A square **is-a-kind-of** shape (uses inheritance).
  - Leroy **is-a** dog (doesn't use inheritance).

- Class Y is like class X except for the following differences…
- The derived class may provide additional ***state*** or ***behavior***, or it may ***override*** the implementation of ***inherited*** methods.

<u>Liskov substitution principle</u>:

- If Y is a subclass of X, then it should be possible to use any instance of Y wherever any instance of X is used.

Questions and Shapes

Imagine a system that asks a series of **questions**:

– YesNoQuestion, NumericQuestion, FreeTextQuestion

- It simplifies things to treat these uniformly, each as a specialization of Question. The program will maintain a list of Questions, and invoke askTheUser( ) for each.

Consider a system that manipulates various kinds of **shapes**:

- Sometimes you don’t care what shape you have (example: move). Sometimes you do care (example: draw).

Polymorphism

"The ability of two or more classes to respond to the same message, each in its own way."

"The ability to use any object which implements a given interface, where the specific class name need not be specified."

Example:

- question.askTheUser();

- To be useful, the responses should be similar in nature.
- Made possible via ***dynamic (run-time) binding***.

Java Example

```java
// File: Derived.java
// What will the following Java code output to the screen?
class Base {
  void foo() { System.out.println( "Base foo" ); }
  void bar() { System.out.println( "Base bar" ); }
}
public class Derived extends Base {
  void bar() { System.out.println( "Derived bar" ); }
  public static void main( String[] args ) {
    Derived d = new Derived();
    d.foo();
    d.bar();
    Base b = d;
    b.bar();
  }
}
```

Modeling

- OO designs begin with an “object model” involving both the domain experts and software designers alike.
- One should model the problem domain and users’ activities.
- The modeling process pins down concepts and creates a shared vocabulary.
- Human thinking about complex situations improves with visual aids.
- A good model is one that shows all the pertinent detail without unnecessary clutter or complexity.
- Who is the audience for your model?
- Learn the Unified Modeling Language (UML).

Example: streets, roads, highways

Classifications depend on the attributes of interest.

- Traffic simulator:
  - one-way, two-way, residential, limited access.
  - location w/ respect to business commuters.

- Maintenance scheduler:
  - surface material.
  - heavy truck traffic.
  - location w/ respect to congressional district.

For every class, say, “This class is responsible for…?”

Example: The “Sticks” Game

- A program is to be written that allows two people to play a game against each other on a computer.
- The game consists of a layout with a number of sticks arranged in rows. When the game starts, they are arranged as shown here:

```
1: |
2: | |
3: | | |
4: | | | |
```

Rules of the Game

- Players alternate turns.
- Players remove one or more sticks from any non-empty row.
- The player who removes the last stick loses.
- At the start of the game, and after each move, the program displays the state of the game, indicates which player is to move, and prompts that player for a row number and the number of sticks to remove from that row.
- The program tells the player when a specified move is invalid, allowing the player to try again.

Find the classes...

Document using CRC cards.

CRC card for Row

The **CRC** approach uses 3x5 index cards, one per **C**lass, which shows its **R**esponsibilities and the other class(es) with which it must **C**ollaborate in order to fulfill each responsibility.

• In this example, class Row must collaborate with class Stick in order to fulfill its responsibility to display itself.

Reducing Complexity

- ***Encapsulation*** exposes only the public interface, thereby hiding implementation details, thus helping to avoid complex interdependencies in the code.
- ***Polymorphism*** allows different classes with the same interface to be interchangeable, making inheritance useful.
- ***Inheritance*** from abstract classes and/or interfaces serves to reduce complexity by allowing ***generalizations***.
- ***Delegation*** reduces complexity by building more complete or higher-level services from smaller, encapsulated ones. Delegation also provides increased run-time flexibility.

Benefits of OO

- Components are good... code reuse.
- Design patterns are good... design reuse.
- Infrastructure and reusable services are also good.
- Interfaces are good... essential to design for loose coupling.
- Interfaces also help to partition human responsibilities.
- Loose coupling and modularity facilitate extensibility, flexibility, scalability and reuse.
- Logical changes are naturally isolated thanks to modularity and information hiding (encapsulation). This leads to faster implementation and easier maintenance.
- OO middleware offers location, platform & language transparencies.
- All of these abstractions are realized in human terms.

Disadvantages of OO

- Slow compared to straight C code.
- Garbage collection can cause slight delays in code execution which can be a disaster in some applications.
- Functional Programming (such as with the Clojure Programming Language) is easier to test and makes side-effects of program code crystal clear.
- The Class is not always the best abstraction to use, nor is it always the most flexible - a lot of JavaScript code opts out of this model.
- OO code can be more verbose than alternatives.