Object- Oriented Programming & Design

Part I: Fundamentals

These notes may be found as PDF files on-line :

www.softwarefederation.com/cs4448.html

What is this course all about?

- Learn the Object-Oriented paradigm.
- Learn the Unified Modeling Language (UML).
- Learn why some designs are better than others.
- Learn how to implement these designs in Java.
- Learn some Design Patterns.
- Learn about 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.

Example: David walks his dog, Leroy

- *Find the objects…*

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.

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 object 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.

These benefits are not automatic.

Design is an art.

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.
- ***Concrete*** 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***?

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.

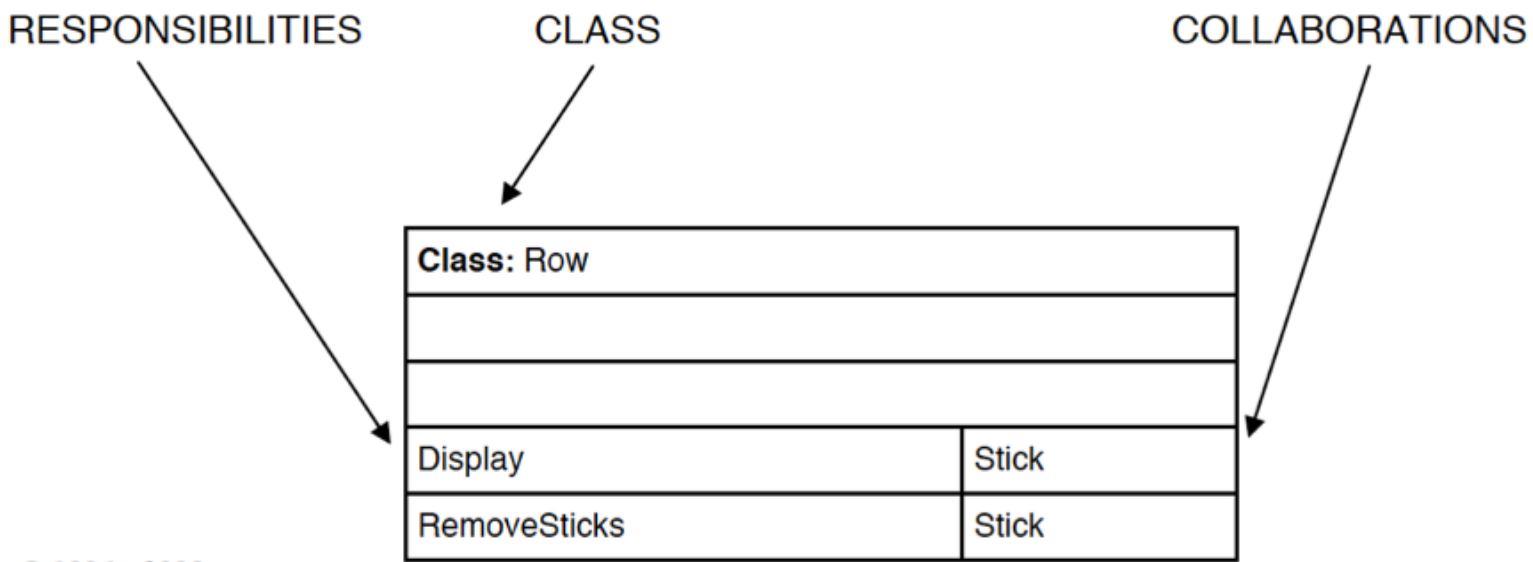
Find the classes...

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.



Responsibility Driven Design

A class' responsibilities define its public interface.

- Walk through the use cases, looking for interactions between objects.
- Where there are interactions, identify client-server relationships.
- Every interaction implies the server has a responsibility.
- Phrase the responsibility from the point of view of the client. This becomes the name of the server's behavior (function, method or operation) that supports the responsibility.
- A large set of responsibilities implies the class should ***delegate*** some work.
- If a client requires a service that does not belong to an existing class, create a new class.

Responsibilities & Collaborations

Responsibilities:

- All the services the instances of a class provide to other objects.
- All instances of a class have the same responsibilities.
- Responsibilities include:
  - Performing actions (methods, behaviors).
  - Maintaining and providing knowledge (state, attributes).
  - Enforcing constraints.

Collaborations:

- Client / Server relationships "in the small."
- A collaboration is one class calling on another (sending a message) to help fulfill a responsibility.

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, increasing team efficiency.
- Loose coupling & modularity facilitate extensibility, flexibility, scalability & 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.

Object - Oriented Programming & Design

Part II: Software Development Process

High-Level Process

- Model the problem domain
  - Develop an ***analysis model***
- Develop ***use cases***
- Model the solution domain
  - Develop a ***design model*** (pass one)
- Implement and test
- ***Iterate…***
  - Within each step, and across all of them
  - Steps are only approximately sequential

Analysis vs. Design

- Analysis describes ***what***
  - An existing business process
  - An existing computing infrastructure
  - The requirements for a solution
- Design describes ***how***
  - A new set of screens
  - A new application
- The distinction is not always clear-cut
  - The customer says “The new system must do x, y and z.”
  - Modeling x, y, and z is analysis
  - Modeling the mechanisms to accomplish x, y, and z is design

Architecture – Design - Implementation

- **Architecture**
  - Hardware, Middleware, Persistence, Networking
  - Programming Language, Frameworks, Infrastructure
  - Scalability, Security, Legacy Systems, TLAs
- **Design**
  - Reduce Complexity, Separate Concerns, Reuse
  - Delegation, Generalization, Interfaces, Layers
  - User Interface, Transactions, Modules, Patterns
- **Implementation**
  - Testing, Data Structures, Algorithms, Thread Safety
  - Encapsulation, Idioms, Memory Management

Model the Problem Domain

- ***Analyze*** the requirements & use cases enough to get started.
- Make a User Interface ***mockup*** with a tool such as Balsamiq.
- Nouns from the problem domain are ***candidate*** objects, classes, and/or attributes.
- For each candidate object, propose a class.
  - Eliminate duplicates (nouns representing the same thing).
  - Eliminate things outside the system that are only clients (e.g., the user).
  - Ignore irrelevant details.
  - An adjective may suggest an attribute or a subclass.
  - Defer or generalize things about which you are uncertain.
- List the verbs: These are candidate responsibilities/behaviors.
- Develop interaction models that support the use case scenarios & system behavior.
- Based on the interaction models, identify class collaborations & relationships.
- Identify responsibilities for classes.
- Break up big, complex classes; consolidate trivial classes.

Model the Solution

- ***Design*** a solution to support what you've analyzed.
- Strive for the simplest solution that will work.
- Avoid over-designing; design only enough for what you know you need.
- Create **UML** models (with pertinent detail).
- Assign operations to classes that will implement the class' responsibilities.
- Identify attributes / associations / inheritance hierarchies.
- Develop an overall system ***architecture***, including solution-domain subsystems.
- Refine existing interaction diagrams and define new ones.
- Study design patterns.
- Ask yourself: What infrastructure is needed in order to make the software easier to extend in the future? Is it easy to build? Will it make things simpler now?
- ***Iterate…*** to develop the simplest solution that will fulfill all of the requirements, while being flexible for future requirements and reuse.
- Realize that you will have more opportunities to refactor later on…

Build and Test

- Implementation involves fine-grained design.
- Implementation issues may suggest or require coarser-grained re-design.
- Design test cases before you code, if possible. Maintain test code.
- Google ***Test-driven development***.
- Write test harness code to drive test cases. In Java, use ***JUnit***.
- Implement a sound ***exception handling*** and ***logging*** strategy right away.
- Post documentation / ***UML*** diagrams to your team's wiki page.
- Use meaningfulVariableNames & write good comments.
- Constantly focus on the overall architecture, adding detail & functionality while ***refactoring*** to simplify & generalize common services & infrastructure.
- Keep clear distinctions between layers & subsystems.
- Strive for simplicity, generalization, reuse & style.

Iterative & Incremental Development

- Analyze and specify only as much as you understand. Defer the rest until you have built what you understand, and understand what you have built.
- Get a working system as soon as possible... but not too fast!
- Be ***architecture-centric***, refactoring design elements as you go along to make them simpler or more flexible, or both.
- Do not be afraid to break working code to ***refactor*** your design if the improved design is worth the effort.
- ***Test*** relentlessly, with a great collection of unit and functional tests.
- Good process helps create better code.
- These activities are interlaced, not strictly sequential.
- Philosophically value working software with continuous user input over thick documents and contracts.
- Avoids analysis paralysis.

Important Guidelines

- Evaluate every decision using principles of good design.
- The first level of organization should be "horizontal" architectural layers and subsystems, *not* "vertical" slices of functionality.
- Each iteration should focus on well-defined and tightly constrained functional objectives (use cases / user stories).
- Implement the basic functionality first, not necessarily the sexiest.
- Don't expect to get it right the first time; a good way to avoid "hacking" is to do it twice (iterate - redesign).
- Be consistent.

For more on Project Management, refer to section XIX.

Refactoring

- Refactoring is the process by which existing portions of code are carefully changed, without adding any new functionality, to become simpler, more general, more flexible, more reusable, easier to understand, easier to maintain, smaller, faster, ...
- Refactoring code involves breaking code that already works, so it is counter-intuitive to people who believe in the old adage "if it works, don't fix it!" Refactoring can be risky! However, it is often worth the effort.
- Refactoring is *disciplined* code evolution. Make small changes, one at a time, to minimize the chance of introducing bugs.
- A good set of unit and functional tests will give you confidence to do more refactoring, with less risk.