Object Oriented Paradigm

Ming-Hwa Wang, Ph.D.
Department of Computer Engineering
Santa Clara University

Object Oriented Paradigm/Programming (OOP)

- similar to Lego, which kids build new toys from assembling the existing construction blocks of different shapes
- OOP - doing programs by reusing (composing/inheriting) objects (instantiated from classes) as much as possible, and only writing code when no exiting objects can be applied

Two Fundamental Principles for OOP

- generalization or reusability: the classes designed should target to reusability, i.e., it should very generic and can be applied to lots of situations/applications without much efforts; reusability comes from inheritance and/or composition
- encapsulation or information hiding: we can hide detailed or implementation specific information away from the users/programmers, so changing implementation using the same interfaces will not need modifying the application code

OOP Language vs. Environment

- a language provides syntax/semantics (by its compiler), e.g., Modular, C++, etc., programmers have to write code for everything to make programs work
- an environment provide a language and its library, e.g., Smalltalk, Java, etc., programmers do not have to write a lot of code to do programming, instead, they just try to reuse the existing code as much as they can

Classes vs. Objects

- a class is a kind or a type, e.g., cat and dog are classes; a class is a data structure, it contains
  - data/attributes/variables to store the states of the class
  - methods/messages/operators/functions/procedures/behaviors to manipulate its data (an implementation), and to provide interfaces/API to the users
    - manager functions manage class objects and handling activities (initialization, assignment, memory management, type conversion), usually invoked implicitly by the compiler, e.g., constructor
    - implementor functions: provide the capabilities associated with the class abstraction
    - helping functions: provide support for the other class member functions, generally declared as private
    - access functions: support user access to otherwise private data
  - level of accessibility: private, protected, public
- an object is an instantiation from a class, e.g., you have a pet cat, who's name is Garfield, then Garfield is a object (of the class cat)
- to invoke an method, simply do obj.method( ) or obj->method( ), the former is object reference syntax, and the latter is pointer syntax

Relationships

- a-kind-of relationship: a subclass inherits and adds more attributes and/or methods from its super class to become somewhat "specialized"
- is-a relationship: an object of a subclass is an object of its super class
- part-of relationship:
- has-a relationship:

Reusability

- inheritance
  - a sub/child/derived/extended class can inherit all properties from its super/parent/base/original class, with or without modification/override or addition
  - single inheritance and multiple inheritance (watch out naming conflicts)
- composition

Abstract or Virtual Classes

- only used as a super class for other classes, only specified but not fully defined, and can't be instantiated
- its derived classes must redefine all the virtual properties

Generic Types: Template

- if a class is parameterized with another type, once an object of that class is created, the parameterized type is replaced by an actual data type

Static and Dynamic Bindings

- static (before compile time) binding
- dynamic (after compile time, i.e., at runtime) binding

Polymorphism

- the ability to request that the same operations be performed by different objects, or many objects to perform the same action
- the concept of dynamic binding allows a variable to take different types dependent on the content at a particular time

Function Overloading

- depending on function name and its parameters (number of parameters and their types, but not return type though)

constructors and destructors

- default constructor or user defined constructors
- copy constructor
- assignment constructor

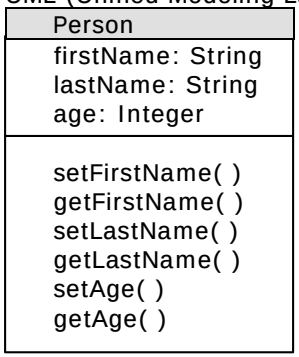
The Design of OOP

the way you analyze a problem (break it down) will give you a particular set of objects, and there are 3 helpful ways to identify objects when you are designing a system:

- a checklist of kinds of objects (Pressman): external entities, things, occurrences or events, roles, organizational units, places, structures
- in the grammatical parse you select the nouns and noun phrases as the potential objects and verbs as possible operations performed on or by the objects
- 6 characteristics to filter a list of potential objects. (Coad Yourdon)
  - retained information - the object needs to remember information
  - needed service - the object has operations which change its attributes
  - common attributes - all occurrences of an object have the attributes
  - common operations - all occurrences of an object have the operations
  - essential requirements - external entities which produce consume information
  - multiple attributes - single attributes might be thought of as being an attribute of a larger object, not an object in their own right

Class Diagrams

- UML (Unified Modeling Language) notation example



← class name

← attributes

← operation

- graphical notation

Design pattern

- the pattern concept: encapsulate change. The basic concept of a pattern can also be seen as the basic concept of program design in general: adding layers of abstraction. Whenever you abstract something, you're isolating particular details, and one of the most compelling motivations for this is to separate things that change from things that stay the same.
- two fundamental design patterns are implemented in object-oriented language compilers: inheritance and composition. Other minor ones are constructors/destructors (as guaranteed initialization and cleanup design pattern), and aggregation (which is erroneously classified as a pattern.)
- Gang of Four (Gamma, Helm, Johnson & Vlissides) discusses 23 patterns, classified under 4 purposes:
  - creational: How an object can be created. This often involves isolating the details of object creation so your code isn't dependent on what types of objects there are and thus doesn't have to be changed when you add a new type of object.
  - structural: These affect the way objects are connected with other objects to ensure that changes in the system don't require changes to those connections. Structural patterns are often dictated by project constraints.
  - behavioral: Objects that handle particular types of actions within a program. These encapsulate processes that you want to perform, such as interpreting a language, fulfilling a request, moving through a sequence (as in an iterator), or implementing an algorithm.
  - concurrency:
- basic ways to keep code simple and straightforward
  - messenger: packages information into an object which is passed around, instead of passing all the pieces around separately
  - collecting parameter: captures information from the function to which it is passed (using container to dynamically add objects)

Name	Description
Creational patterns	
Abstract factory	Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Factory method	Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Builder	Separate the construction of a complex object from its representation so that the same construction process can create different representations.
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed.
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
Singleton	Ensure a class only has one instance, and provide a global point of access to it (using static member and making all constructors/destructors private).
Utility	A class with a private constructor that contains only static methods.
Structural patterns	

Adapter	Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
Bridge	Decouple an abstraction from its implementation so that the two can vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of fine-grained objects efficiently.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Behavioral patterns	
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations. An object-oriented replacement for callbacks (also decoupling event handling).
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
Observer	Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Specification	Recombinable business logic in a boolean fashion
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
Single-serving visitor	Optimize the implementation of a visitor that is allocated, used only once, and then deleted
Hierarchical visitor	Provide a way to visit every node in a hierarchical data structure such as a tree.
Concurrency patterns	
Active Object	The Active Object design pattern decouples method execution from method invocation that reside in their own thread of control. The goal is to introduce concurrency, by using asynchronous method invocation and a scheduler for handling requests.
Balking	The Balking pattern is a software design pattern that only executes an action on an object when the object is in a particular state.
Double checked locking	Double-checked locking is a software design pattern also known as "double-checked locking optimization". The pattern is designed to reduce the overhead of acquiring a lock by first testing the locking criterion (the 'lock hint') in an unsafe manner; only if that succeeds does the actual lock proceed. The pattern, when implemented in some language/hardware combinations, can be unsafe. It can therefore sometimes be considered to be an anti-pattern.
Guarded	In concurrent programming, guarded suspension is a software design pattern for managing operations that

	require both a lock to be acquired and a precondition to be satisfied before the operation can be executed.
Leaders/followers	
Monitor object	A monitor is an approach to synchronize two or more computer tasks that use a shared resource, usually a hardware device or a set of variables.
Read write lock	A read/write lock pattern or simply RWL is a software design pattern that allows concurrent read access to an object but requires exclusive access for write operations.
Scheduler	The scheduler pattern is a software design pattern. It is a concurrency pattern used to explicitly control when threads may execute single-threaded code.
Thread pool	In the thread pool pattern in programming, a number of threads are created to perform a number of tasks, which are usually organized in a queue. Typically, there are many more tasks than threads.
Thread-specific storage	Thread-local storage (TLS) is a computer programming method that uses static or global memory local to a thread.
Reactor	The reactor design pattern is a concurrent programming pattern for handling service requests delivered concurrently to a service handler by one or more inputs. The service handler then demultiplexes the incoming requests and dispatches them synchronously to the associated request handlers.