Chapter 1: Software and Software Engineering

- Cost is mostly in development, easy to reproduce
- Software doesn't 'wear out' *deteriorates* by design changed
- Much software has poor design and is getting worse
- Term 'Software Engineering' was coined in 1968
- SE Code of Ethics:
  - ○Act consistently with public interest
  - ○Act in the best interests of their clients
  - ○Maintain integrity and independence
  - ○Promote an ethical approach in management
  - ○Advance integrity and rep. Of profession
  - ○Be fair and supportive to colleagues
  - ○Participate in lifelong learning
- ●Software quality:
  - ○**Usability**: users can learn it fast and get their job done easily
  - ○**Efficiency**: doesn't waste resources (CPU time and mem.)
  - ○**Reliability**: low fail %
  - ○**Maintainability**: can be easily changed
  - ○**Reusability**: parts can be used in other projects
- ●Most projects *evolutionary/maintenance* projects legacy systems)
  - ○Corrective: Fixing Defects
  - ○Adaptive: changing system in response to operating system, database, rules and regulations
  - ○Enhancement: adding new features
  - ○Reengineering/perfective: making system more maintainable

Chapter 2: Review of Object Orientation

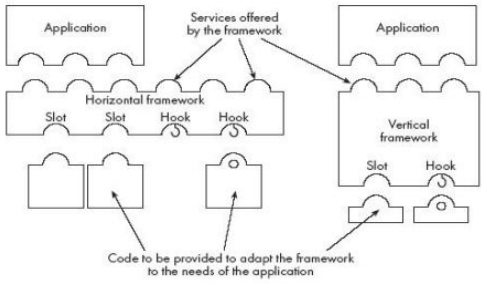
- ●Procedural Paradigm:
  - ○Organized around the notion of *procedures*
  - ○*Procedural abstraction:* works as long as the data is simple
- ●Adding data abstractions groups together the pieces of data that describe some entity; reduces system's complexity
- ●Object Oriented Paradigm
  - ○Organizing procedural abstractions in the context of data abstractions
- ●Running program can be seen as a collection of objects collaborating to perform a task

Procedural vs OO

- ●Object:
  - ○Chunk of structured data in a software system
  - ○Has properties (represents its state)
  - ○Has behaviour (how it acts/reacts, simulates behaviour in reality)
- ●Class:
  - ○Unit of abstraction in OO program
  - ○Represents similar objects (its instances)
  - ○Software module (describes instances properties, contains methods to implement behaviour)
- ●Class vs Instance:
  - ○Class: Film, Reel of Film, Science Fiction Film
  - ○Instance: Film with serial no. SW19876, Science Fiction
- ●Instance Variables:
  - ○fields/member variables
  - ○Attributes (simple data, e.g. name, DoB)
  - ○Associations (relationships to other classes)
- ●Variable refers to an object, can point to different objects in time
- ●Object can be referred to by multiple variables
- ●Class variable: shared by all instances (static var) (e.g. constants)
- ●Operation: high-level procedural abstraction, specifies behaviour
  - ○Independent of code which implements that behaviour (area)
- ● Method: procedural abstraction used to implement behaviour
  - ○Several classes can have methods with the same name (calculating area in a rectangle vs circle)
- ●Polymorphism: Property of OO software which an abstract operation may be performed in different ways in different classes
  - ○Multiple methods of the same name
  - ○Choice to execute depends on the object in variable
- ●Inheritance Hierarchies
  - ○Superclasses: contains common features to subclasses
  - ○Inheritance Hierarchies: shows relationship between super/sub
  - ○Inheritances: implicit possession by subclasses features in super
- ●Is-a rule: check generalizations (a checking account *is an* account)
- ●Abstract class: Cannot instantiate this class (e.g. Shape)
  - ○Leaf classes must have/inherit concrete methods of super
- ●Method precedence: current class first, check immediate super
- ●Dynamic binding: occurs when decision about which method to run can only be made at run time
- ●Key Terminology:
  - ○**Object** -> Something in the world
  - ○**Class** -> Objects
  - ○**Superclass** -> subclasses
  - ○**Operation** -> methods
  - ○**Attributes and associations** -> instance variables
- ●Modularity: Code divided into classes, classes into methods
- ●Encapsulation:
  - ○Details can be hidden in classes; information hiding

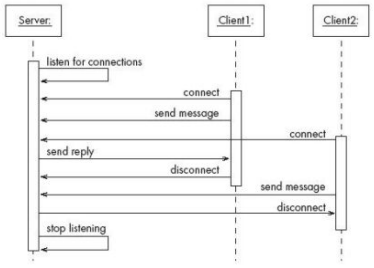
Chapter 3: Basing Soft Dev on Reusable Tech

- ●Framework: reusable software that implements generic solution to generalized problem; *not complete Application*
- ●Principle: Applications that do different, but related things tend to have similar designs
- ●Certain classes or methods used by framework, but missing slots that are provided by application
- ●In OO, framework composed of library of classes, API defines set of all public methods for these classes (abstract)
- ●Product line: set of products built on common tech base, software common to all included on framework,
  - ○Product produced by filling available hooks and slots



Types of framework

- ●Types of Framework:
  - ○Horizontal: provides general app facilities (general)
  - ○Vertical: more 'complete' but still needs some slots filled
- ●Client-Server: distributed system; computations are cooperated by separate programs on different hardware
  - ○Server: program that provides service for connected prog.
  - ○Client: program that accesses a server for services

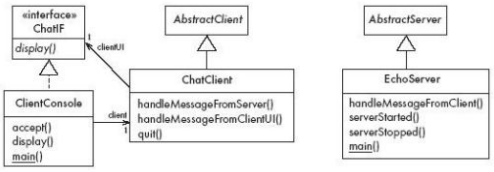


Typical life cycle of a server/client

- ●Fat vs. thin clients:
  - ○Fat: As much work as possible delegated to client, server can handle more clients
  - ○Thin: Client made as small as possible, most work done on server, easy to download on network
- ●IP (Internet protocol): long messages are split up into small packets, connectionless protocol, route messages

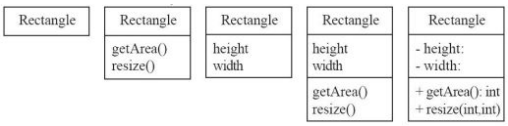
- ***●Object Client Server Framework (OCSF)***

- ●Framework never supposed to be modified; use as library to decouple from future changes
- ●To properly use: Develop subclasses provided, call public methods provided, override slots/hooks as needed
- ●Client side:
  - ○Single class AbstractClient must be extended in normal class, implement handleMessageFromServer, takes hostname and port number as argument
  - ○Status Access methods
    - ■isConnected() Hook
    - ■getPort()/setPort(int Port)
    - ■getHost()/setHost(String host)
    - ■getInetAddress()
  - ○Callback methods
    - ■ConnectionEstablished()
    - ■ConnectionClosed()
  - ○Instance Variables:
    - ■Host, port: define server and port to use
    - ■Client socket: abstract all info about connection
    - ■Input/output streams handles the coding/buffering of messages passed between client/server
    - ■Thread that implements run method
- ●Server side:
  - ○Two main classes:
    - ■AbstractServer: listens for new connections
    - ■ConnectionToClient: Handle connection to clients (1 per)
  - ○Server instantiation
    - ■AbstractServer(int port)
  - ○Server functionality
    - ■listen(), stopListening()
    - ■close()
    - ■sendToAllClients()
    - ■receiveMessageFromClient()
  - ○Callback functions
    - ■serverStarted/Stopped()
    - ■clientConnected/Disconnected/Exception() Hook
    - ■handleMessageFromClient() - (must be overridden)
  - ○Status access function
    - ■isListening()
    - ■getNumberOfClients()
    - ■setClientConnections()
    - ■getPort()
    - ■setPort()
  - ○Instance Variables:
    - ■Port
    - ■Collection of instances of ConnectionToClient is stored using a special class called ThreadGroup
    - ■Timeout to pause listening to check if close connection was requested

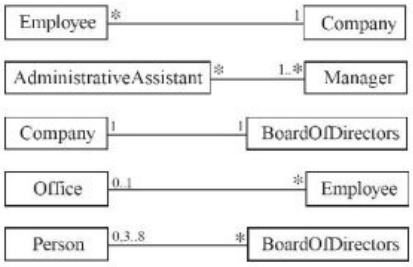


Chapter 5: Modeling with Classes

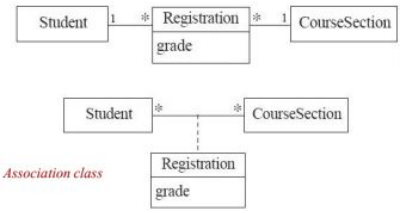
- Classes: represent the types of data themselves
- Associations represent linkages between instances
- Attributes are simple data found in classes and inst.
- Operations represent abstract functions performed by classes and instances, as well as specific methods implementing these
- Generalizations group classes into inheritance hierarchies
- **Classes** in UML
  - Represented as a box with the name of the class inside, with optional attributes and operations



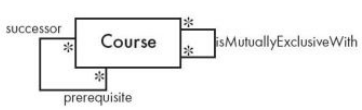
- **Association** used to show how classes relate to each other, symbols indicate multiplicity at ends



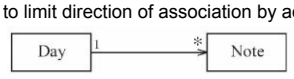
- Types of Multiplicity:
  - Many-to-One: ex. A company has employees, an employee can work for only one company
  - Many to many: ex. An assistant can work for many managers, a manager can have many assistants
  - One to one: A company can only have 1 board of directors, a board of directors is only for 1 company
- Association Classes
  - Sometimes, attribute that concerns two associated classes cannot be placed in either



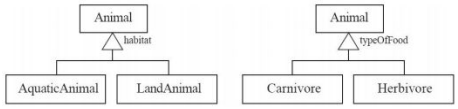
- Reflexive Association: possible to connect class to itself
  - asymmetric (multiple associations link back up) and symmetric (only one, associations linking back to itself). The lower bound should always be 0.



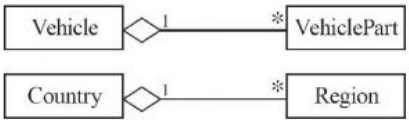
- Directional Associations: bi-directional by default, possible to limit direction of association by adding arrow



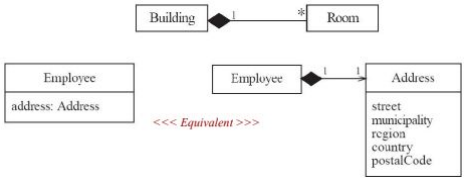
- Class Generalization
  - Generalization set is a labeled group of generalizations with a common superclass, label describes criteria used



- Associations vs Generalization in object diagrams
  - Associations describe relations that will exist between *instances* at run time (there will be an instance of both classes joined by association)
  - Generalizations describe relationships between classes in class diagrams (an instance of any class should be considered an instance of the superclass)
- Aggregations: represent 'part-whole' relationships; whole is known as the aggregate, (isPartOf)
  - Parts can exist without the whole

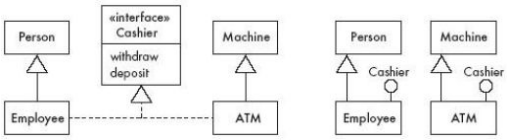


- Composition: strong aggregation; if aggregate is destroyed, the parts are destroyed as well



- Propagation: operation in an aggregate is implemented by having the aggregate perform that operation on its parts
  - Properties of parts are propagated back to the aggregate
  - Propagation is to aggregation as inheritance is to generalization

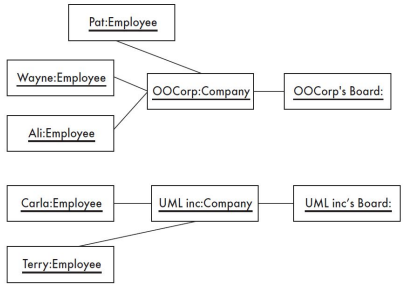
- Interface: describes portion of visible behaviour of set of objects, similar to class, except lacks instance variables and implementation



- **Developing Class Diagrams:**
- Class Diagram Level of Detail
  - Exploratory domain model: learn about the domain
  - System domain model: model aspects of domain
  - System Model: includes classes used to build user interface and system architecture

Object Diagrams:

- AKA instance diagrams, shows a configuration of objects and links that may exist at one point during execution of a program.
- Objects are represented as rectangles and their names are underlined, have a colon them and can be given names (ex: Forward:Professor).
- Links between objects are lines and WE DO NOT PUT MULTIPLICITIES ON LINKS.An object diagram is generated by a class diagram.
  - This means that it contains instances and links of the classes and associations present in the class diagram.
  - It also means that the number of links among instances are consistent with the multiplicity of that class diagram. A class diagram can generate an infinite amount of multiplicities. 2 examples:



Common Mistakes:

- unnecessary generalizations: not using the three rules of inheritance;
  - (1) is-a rule which means that all subclasses can be put in the form "A is a(n) B" where A is the subclass and B is the superclass (basically it makes sense in english using "is a"),
  - (2) subclasses should have unique attributes or associations to them,
  - (3) all subclasses should have common attributes but not all.

- incorrect location of attributes in generalizations & creating a class when you should be using an attribute:
  - this is when an attribute is falsely used a class on it's own. It's crucial to determine whether a specification of the system requirements is a class or an attribute of a class.
- Incorrect usage of Aggregation and reflective associations in UML:
  - for example if we have a class order for ordering food. Let's also say that we want to keep track of orders made in a log. Having a reflective association with order isn't correct.
  - Because there is no such thing as suborders. The more suitable representation is having a third class that has a one to many association with class order.
- using arrow or multiplicities in instance/object diagrams:
  - In object diagrams we don't have multiplicities or associations only links. Hence, when illustrating an object diagram it's important to keep note that each instance of a class (object) should be linked to other instances.
  - for example if we had in a UMl diagram a 1 to many associations, in the object diagram we should have the object be linked with several objects that are instances of classes.
- Not underlying the instances in object diagrams:
  - The professor stressed on this numerous times. While building the object diagram do not forget to underline the instance name and class name of any given object.
- Irrelevant assumptions :"
  - Assumptions should be relevant to the question and design. I saw many students make assumptions without any bases." -The Professor Miguel Garzón. This probably means that the assumptions that the students made where due to lack of understanding of the UML and them wanting to rectify those mistakes.

Previous midterm example: